Institut für Informatik

Dissertation

Describing Differences between Overlapping Databases

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (doctor rerum naturalium)

eingereicht an der
Mathematisch-Naturwissenschaftliche Fakultät
der Humboldt-Universität zu Berlin

von
Diplom-Informatiker Heiko Müller,
geboren am 29. Oktober 1970 in Berlin

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies
Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter: 1. Prof. Johann-Christoph Freytag, Ph.D.

2. Prof. Ulf Leser

3. Prof. Bertram Ludäscher

Datum der Einreichung: 22. April 2008

Datum der Promotion: 19. Dezember 2008

Acknowledgements

I thank my advisors Prof. Johann-Christoph Freytag and Prof. Ulf Leser for their support, their helpful advice, and the time they spend with me in informative discussions during the preparation of this thesis. I especially want to thank Prof. Freytag for his always open words and the patience he had with me during my work. His database department at the Humboldt University Berlin provided a really inspiring research environment for me. Prof. Leser was the one who created the idea of writing this thesis in me. His valuable and challenging comments throughout my work significantly influenced the topic of this thesis and helped to improve its final outcome.

This work was financially supported by the Berlin and Brandenburg Graduate School for Distributed Information Systems. The graduate School with its regular workshops and evaluation by its professors provided a challenging, but supportive and productive environment for me and my research. I would like to thank all my colleagues and all professors for their comments throughout the years.

I discussed my work with many colleagues at Humboldt University and the database group. I thank Prof. Felix Naumann for his support and for introducing me to the problems and challenges of data quality. I also thank Dr. Stephan Heymann and Peter Rieger for sharing their knowledge about genome research and quality pitfalls of genome data. I would like to thank Heinz Werner and Thomas Morgenstern for their excellent technical support and Ulrike Scholz for her administrative support. Finally, a big ‘Thank You’ to all my colleagues during the years at he database group who made my time an enjoyable one.

Outside of the academic world, I want to thank my family and friends for their support. I assume it hasn’t always been easy to ‘endure’ me during that time. I dedicate this work to my father and my grand mother, who both were very proud to see me working on this thesis, but who both are no longer with us to witness the final moments.

Zusammenfassung

Die Analyse existierender Daten ist wichtiger Bestandteil vieler Forschungsaktivitäten. Insbesondere im Bereich der medizinischen und pharmazeutischen Forschung entscheiden die Ergebnisse dabei nicht nur über eine erfolgversprechende Verwendung finanzieller Mittel, sondern oftmals auch über das Wohlergehen von Probanden und Patienten. Analysen die auf der Grundlage von fehler- oder mangelhaften Daten durchgeführt werden können deshalb schwerwiegende negative Folgen haben. Aus diesem Grund hat das Thema Datenqualität im Bereich der wissenschaftlichen Forschung in den vergangenen Jahren zunehmend an Bedeutung und Aufmerksamkeit gewonnen. Existierende regelbasierte Verfahren zur Qualitätskontrolle und Datenbereinigung sind für wissenschaftliche Daten jedoch nur bedingt einsetzbar. Dies liegt zum einen an der höheren Komplexität der Daten und zum anderen an unserer oftmals noch unvollständigen und mit Unsicherheit behaftet Kenntnis der Regularien in den entsprechenden Domänen. Die vorliegende Arbeit ist in drei Teile gegliedert und leistet folgende Beiträge im Hinblick auf Datenqualität und Datenbereinigung in wissenschaftlichen Datensammlungen:

Im **ersten Teil** der Arbeit geben wir einen Überblick über existierende Verfahren zur Datenbereinigung und diskutieren deren Stärken und Schwächen hinsichtlich der Beseitigung von Qualitätsproblemen in wissenschaftlichen Daten.

Wir beginnen mit einer Klassifikation von Unzulänglichkeiten in existierenden Datenbanken, die zu einer Minderung der Datenqualität führen. Datenqualität wird generell als Vektor unterschiedlicher Qualitätskriterien definiert. Für jede der definierten Problemklassen geben wir die Qualitätskriterien an, die von diesen Problemen negativ beeinträchtigt werden. Auf Grundlage dieser Zuordnung geben wir einen Überblick über existierende Ansätze zur Bereinigung von Daten und zeigen auf, welche Qualitätskriterien von welchen Ansätzen bedient werden. Aus unseren Ergebnissen folgern wir, daß überlappende Datenquellen großes Potential hinsichtlich Verbesserung der Korrektheit und Genauigkeit von Daten haben. Der vergleich überlappender Datenquellen deckt Bereiche potentiell minderer Datenqualität in Form von Datenkonflikten auf. Gleichzeitig bieten die überlappenden Daten eine Möglichkeit zur Qualitätsverbesserung durch Datenintegration.

Am Beispiel von Genomdaten zeigen wir, daß Datenqualitätsprobleme in wissenschaftlichen Daten zum großen Teil im Produktionsprozeß der Daten begründet sind. Wir analysieren den Produktionsprozeß und identifizieren verschiedene Formen von Qualitätsproblemen und deren Verursacher. Da eine manuelle Qualitätskontrolle während der Datengenerierung aus Effizienzgründen nicht praktikabel ist, muß eine Datenbereinigung a posteriori vorgenommen werden. Anhand praktischer Arbeiten diskutieren wir die Vor- und Nachteile unterschiedlicher Ansätze. Die Integration überlappender Datenquellen stellt besonders in diesem Bereich einen vielversprechenden Ansatz dar.

Eine wichtige Voraussetzung für die Integration überlappender Datenquellen besteht in einem gezielten Auflösen der auftretenden Datenkonflikte (kurz Konflikte). Aus einer Menge an widersprüchlichen Werten gilt es den oder die zuverlässigsten Werte auszuwählen und daraus einen sog. Repräsentanten abzuleiten. In vielen Fällen treten die Konflikte nicht zufällig auf sondern folgen einer systematischen Ursache. Eine Kenntnis dieser Systematik erlaubt es Konflikte mit gleicher Ursache gemeinsam zu lösen. Wir bezeichnen dies als kontextabhängige Konfliktlösung. Im **zweiten Teil** dieser Arbeit entwickeln wir eine Reihe von Algorithmen, die das Auffinden von systematischen Unterschieden in überlappenden Daten unterstützen.

Wir präsentieren ein Modell für systematische Konflikte in überlappenden Daten. Wir klassifizieren Konflikte dabei anhand charakteristischer Muster in den überlappenden Daten, die im Zusammenhang mit diesen Konflikten auftreten. Diese Widerspruchsmuster dienen einem Experten als Unterstützung bei der Festlegung von Konfliktlösungsstrategien im Rahmen der Datenintegration. Widerspruchsmuster stellen eine spezielle Form von Assoziationsregeln dar. Basierend auf existierenden Techniken präsentieren wir effiziente Algorithmen zur Suche nach Widerspruchsmustern in überlappenden Datenquellen. Um die Vielzahl der potentiellen Widerspruchsmuster handhaben zu können definieren wir verschiedene Maße für deren Relevanz. In unseren Experimenten diskutieren wir den Einfluß dieser Maße auf die Aussagekraft und die Anzahl der gefundenen Widerspruchsmuster.

Widerspruchsmuster sind hilfreich bei der Identifikation von Konflikten, die eine gemeinsame Konfliktursache haben. Im **dritten Teil** dieser Arbeit verwenden wir ein prozeßbezogenes Model zur Beschreibung systematischer Konflikte, um Abhängigkeiten zwischen Konfliktgruppen aufzeigen zu können.

Wir verwenden hierzu Sequenzen mengenorientierter Modifikationsoperationen die eine Datenquelle in die andere überführen. Jede Sequenz die eine Datenquelle in die andere überführt muß sämtliche Konflikte zwischen den Quellen auflösen. Die minimale Sequenz hinsichtlich der Anzahl an Operationen ist die kleinstmögliche Zusammenfassung sämtlicher Unterschiede zwischen den Datenquellen. Wir präsentieren Algorithmen zur Bestimmung minimaler Modifikationssequenzen für ein gegebenes Paar von Datenquellen. Die Komplexität des Problems bedingt die Verwendung von Heuristiken für große Datensätze. Wir präsentieren eine Reihe solcher Heuristiken, die jedoch nicht immer die optimale (sprich minimale) Lösung finden. In unseren Experimenten zeigen wir, daß die Qualität der Ergebnisse unserer Heuristiken dennoch sehr vielversprechend ist.

Die in dieser Arbeit präsentierten Widerspruchsmuster und Modifikationssequenzen helfen systematische Unterschiede zwischen überlappenden Datenquellen aufzudecken. Unsere Algorithmen liefern somit wertvolle Informationen zur qualitativen Bewertung überlappender Daten. Die Ergebnisse können sowohl zur Spezifikation von Konfliktlösungsstrategien als auch bei der Verbesserung des Datenproduktionsprozeß eingesetzt werden. Diese Arbeit bildet daher eine wohl fundierte Basis zur kontextbasierten Konfliktlösung und zur Steigerung der Qualität wissenschaftlicher Daten.

Abstract

High costs and loss of reputation caused by data of poor quality made quality assurance and data cleansing hot topics in the business world over the past decades. Recently, data quality has become an issue in scientific research as well. Cleaning scientific data, however, is hampered by incomplete or fuzzy knowledge of regularities in the examined domain. Thus, we are limited in our ability to specify a comprehensive set of integrity constraints to assist in identification of erroneous data. For this reason, overlapping databases are becoming the primary source of information for detecting hot-spots of poor data quality and for data cleansing. A common approach to enhance the overall quality of scientific data is to merge overlapping sources by eliminating conflicts that exist between them to form a single high-quality data set.

Overlapping databases are valuable sources of information for data cleansing, provided that we are able to identify and resolve differences effectively. Deciding on what value is to be taken from a given set of conflicting values or how a solution is to be computed requires input from an expert user familiar with domain constraints, regularities, and possible pitfalls in the data generation process. However, high numbers of conflicts between overlapping databases makes manual inspection of individual conflicts infeasible. The main objective of this thesis is to provide methods to aid the developer of an integrated system over contradicting databases in the task of resolving value conflicts. We contribute by developing a set of algorithms to identify regularities in overlapping databases that occur in conjunction with conflicts between them. These regularities highlight systematic differences between the databases. Evaluated by an expert user the discovered regularities provide insights on possible conflict reasons and help assess the quality of inconsistent values. Instead of inspecting individual conflicts, the expert user is now enabled to specify a conflict resolution strategy based on known groups of conflicts that share the same conflict reason.

The thesis has three main parts. **Part I** gives a comprehensive review of existing data cleansing methods. We classify data deficiencies that diminish the quality of existing data sources and quality criteria that are affected by these deficiencies. Based on these classifications, we show which cleansing approaches are capable of handling which data deficiencies and quality criteria. We show why existing data cleansing techniques fall short for the domain of genome data and argue that merging overlapping data has outstanding ability to increase data accuracy; a quality criteria ignored by most of the existing cleansing approaches.

Part II introduces the concept of contradiction patterns. We present a model for systematic conflicts and describe algorithms for efficiently detecting patterns that summarize characteristic data properties for conflict occurrence. These patterns help in providing answers to questions like “Which are the

conflict-causing attributes, values, or value pairs?" and "What kind of dependencies exists between the occurrences of contradictions in different attributes?".

Contradiction patterns define classes of conflicts that potentially follow the same conflict reason. Contradiction patterns, however, cannot reveal any dependencies regarding the origin of conflicts during data generation. In **Part III**, we define a model for systematic conflicts by using update operations. Sequences of set-oriented update operations are used as abstract descriptions for regular differences among databases. Given a pair of contradicting databases, each operation may (i) represent an update that has been performed on one of the databases (considering that both have evolved from a common ancestor), or (ii) describe systematic differences in their respective data production processes. Update sequences give valuable insights why a database is different from its original state. Even though we only consider a restricted form of updates, our algorithms for computing minimal update sequences for pairs of databases require exponential space and time. We show that the problem is NP-hard for a restricted set of operations. However, we also present heuristics that lead to convincing results in all examples we considered.

Contents

Part I

Data Quality and Data Cleansing

Chapter 1

Introduction

The decreasing cost for generating and maintaining scientific data has led to an enormous increase in the number of scientific data sources available on the Internet today. The increase is, for example, reflected by the number of biological data collections listed in the NAR MOLECULAR BIOLOGY DATABASE COLLECTION that rose from 202 entries in 1999 [Bur99] to 968 entries in 2007 [Gal07]. There are various examples for different biological data sources that overlap in the set of objects they represent. Two frequent scenarios lead to such overlap:

- **Data Replication**: A common example for overlaps between scientific data sources is the set of three databases GENBANK [BKM+07], EMBL [KAA+07], and DDBJ [OSGT06] within the INTERNATIONAL NUCLEOTIDE SEQUENCE DATABASE COLLABORATION (INSDC) [INSDC]. These databases all manage the same set of DNA sequences, but share the burden of submission handling and query answering. Data from the INSDC databases is used as the basis for many genome research projects and is therefore copied to numerous other databases. For example, the ALTERNATIVE SPLICING DATABASE (ASD) [TSC+04], the EXTENDED ALTERNATIVELY SPLICED EST DATABASE (EASED) [PHBR04], and SPLICENEST [KHCV02] all replicate Expressed Sequence Tag (EST) data from the INSDC databases as input for their operational pipelines to predict alternative splice forms in gene expression.
- **Different groups administrating, analyzing, or observing the same set of real-world objects**: A common practice in scientific research is to distribute the same set of samples, such as clones, proteins, or patient's blood, to different laboratories for analysis to enhance the reliability of the final results. In other scenarios overlapping data is generated independently by different groups due to research projects that focus on the same organism, disease, or metabolic process. For example, from the 46 Prokaryotic genome data sources listed in [Gal07] eleven are devoted to *Escherichia coli*, each maintaining various annotations for the genome of the organism.

In general, we distinguish between controlled and uncontrolled data overlaps. Replication of data on different machines for load balancing or for security reasons is a common example for controlled overlap maintenance. Uncontrolled overlaps occur, for example, when data is copied from web sources or data is produced independently at different locations. Whenever data is distributed or generated without a control scheme enforcing consistency, there is a high probability that actual values will differ in the different data sets. Reasons might be modification or transformation of replicated data copies, filtering or errors in the replication mechanism, different levels of actuality of the data, or imprecision of

measurement and systematic bias in data generation. Differences may also result from data cleansing operations being performed independently on the replicas. In any case, different values result in inconsistencies between the overlapping data sources due to contradicting representations for the same real-world object or fact. An integration system trying to generate a consistent view of the data faces several problems. First, it must identify inconsistencies in an efficient manner. Second, it must resolve these inconsistencies, e.g., by selecting a certain value due to a quality score assigned to the data sources or the conflicting values themselves. Third, the system will be interested in finding the source of the deviations to avoid such problems in the future. While the later is not primarily an integration problem, having knowledge about the sources for deviations is helpful not only for future conflict avoidance but also for assessing the quality of conflicting values for conflict resolution.

The main objective of this thesis is to provide methods to aid the developer of an integrated systems in the second and the third task. We contribute by developing a set of algorithms to identify regularities in overlapping data sources that occur in conjunction with contradictions between them. These regularities highlight systematic differences between the data sources. Evaluated by an expert user the discovered regularities provide insights on possible conflict reasons and help assess the quality of inconsistent values. This information is valuable for quality assessment of the different data sources and in implementing a conflict resolution strategy.

1.1 Merging Overlapping Data Sources

Contradicting data, in general, is objectionable due to the need for conflict resolution. There are, however, situations where overlapping sources provide the benefit of offering different opinions or views. Especially in scientific research overlaps are utilized to identify potentially erroneous data as a basis to improve data quality [MWBL05]. Scientific data results from experiments and analysis of experimental results. Quality of the data is dependent on the experimental setup, the reliability of the used equipment, and the expertise of the operator. While data has become of great importance for many research efforts the quality of the data sources at hand is considered to be doubtful in many cases. Cleaning scientific data is hampered by incomplete or fuzzy knowledge of regularities in the examined domain limiting the ability to specify a comprehensive set of integrity constraints to assist in identification of erroneous data. Due to this restriction, comparing overlapping data sources has become the primary source of information for detecting hot-spots of poor data quality and for data cleansing.

Example 1-1: In 1999 *Steven E. Brenner* compared functional annotations for the *Mycoplasma genitalium* genome, performed independently by three different groups, to estimate the accuracy of automatic functional annotation [Bre99]. His results show that of the reported 468 genes only 340 are annotated by two or more groups. For about 8% of these 340 related functional annotations the descriptions of at least two of the groups are completely incompatible. ■

A common approach to enhance the overall quality of scientific data is to merge overlapping sources by eliminating conflicts that exist between them. As a result, a single, high-quality data set is formed. The HUMAN GENOME PROJECT [IHGSC04, VAM+01] is a typical example for the same set of clones being sequenced multiple times. The individual results are merged to form the consensus sequence of the human genome. In order to achieve the targeted exactness of 99.99% for the determined sequence, each base is sequenced six times on average [DG02].

Generating a consistent view on a set of contradicting data sources is considered a special case of data integration, referred to as *data merging*. In general, data integration is defined as the problem of combining data from different sources to provide the user with a unified and comprehensive view of the data [Len02]. The need for data integration exists ever since information is been spread across heterogeneous data sources. The NAR MOLECULAR BIOLOGY DATABASE COLLECTION currently list a total of 244 different data sources containing information about protein sequences, sequence motifs and families, protein structures, and the involvement of proteins in metabolic and signaling pathways. In order to generate a comprehensive view on the sequence, the properties, and the process involvement of a particular protein, a biologist needs to browse and combine data from a large number of sources. Manual integration of information is tedious. Each data source is usually searchable via a proprietary interface having varying search capabilities. Search results are often represented in a source specific format. Querying such data sources requires to copy and paste data between the various interfaces and manually transform data into different formats. An integration system hides any heterogeneity in the data models, query interfaces, schemas, and value formats of the sources from the user by defining a unified view of the data, called the global schema. The global schema is a reconciled view of the information in the data sources that can be queried by the user. Queries are formulated against the global schema and the integration system is responsible for retrieval and transformation of relevant data from the sources to answer the query. Correspondences between data sources and the global schema are specified by schema mappings. These mappings describe how data structured under the source schema is to be transformed into data structured under the global schema by a set of data transformations. *Sheth and Larson* give a systematic overview of data integration systems [SL90].

Rather than enhancing the completeness of data by combining complementing information from different sources, the primary objective in data merging is to enhance *data consistency* and *correctness*. There are three main steps in data merging, as outlined in Figure 1-1: (i) schema mapping and transformation, (ii) duplicate detection, and (iii) conflict resolution. These steps are common to any integration effort featuring overlapping sources.

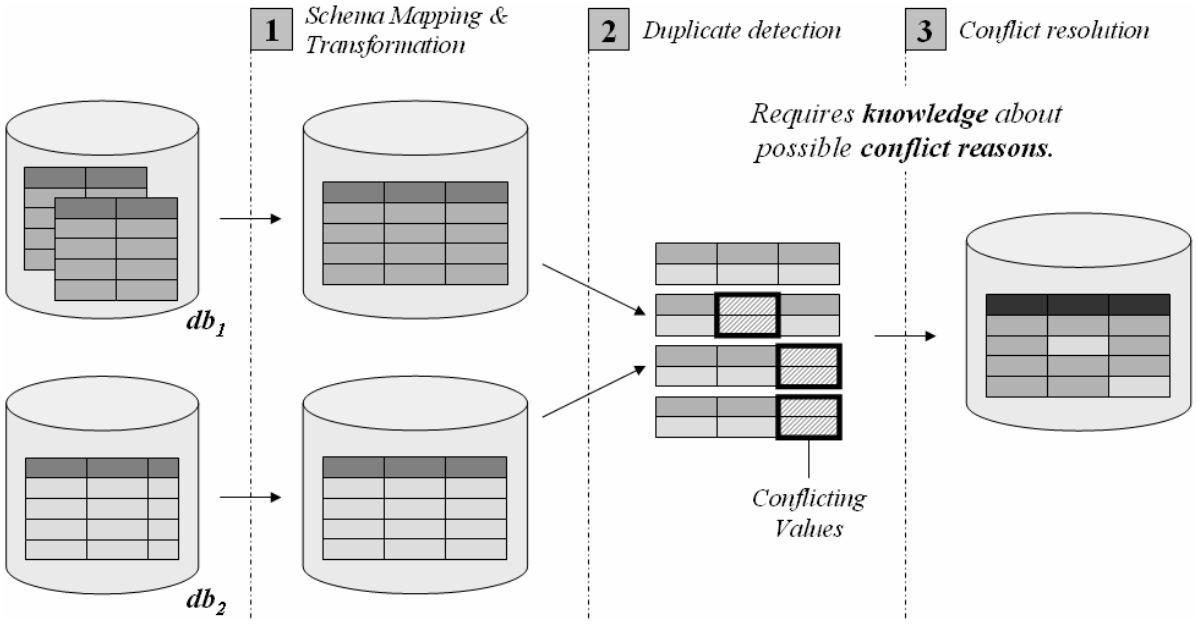


Figure 1-1: An overview the data merging process for overlapping sources consisting of (1) schema mapping and transformation, (2) duplicate detection, and (3) conflict resolution.

In the first step, overlaps between schemas of the sources and the global schema are detected. In general, there are two different ways for data sources to overlap: in their intension and in their extension. The *extensional overlap* between two sources is the set of real-world objects that are represented in both sources. The *intension overlap* between two sources is the set of entity types and attributes that both sources provide [Nau02]. In this thesis, we assume that data sources overlap completely with the global schema, i.e., each source contains information about the same set of properties for the represented real-world objects. Intension overlaps are specified as mappings between attributes in the sources and the global schema. Data merging resembles a local-as-view approach of data integration, where the sources are characterized as views over the global schema. Definition of these views is based on the identified schema mappings. The result of the mapping and transformation phase is a set of data sources structured under the global schema.

The second step of the merging process identifies overlaps in the extension of the data sources. This step is referred to as *duplicate detection*. Duplicate detection is complex in the presence of inconsistencies and in the absence of a source-spanning global object identifier. The main challenge is to decide whether two records with non-identical values are considered representations of the same real-world object. Therefore, duplicate detection depends on a similarity measure for data records. The most common similarity measures are either based on measuring the similarity of conflicting values or on similarity in the relationships of a record with other records (see [EIV07, Win99] for surveys on duplicate detection methods). Duplicate entries may also exist within a single source, thus making steps two and three of the data merging process also applicable to generate a consistent view of a single data source. The result of duplicate detection is an assignment of global unique object identifiers for each record that enable efficient identification of duplicate object representations.

In the final step of data merging, a single consistent representation of the data is generated. This step is called *conflict resolution*. For each set of duplicate records a single representative is derived. Whenever duplicate records disagree on an attribute value a single representative value is derived from the existing ones. The final result of data merging is a consistent dataset where each record from the initial sources is represented by a single unified entry. Records that do not share duplicates in any of the sources are simply passed through in the third step. In the following, we consider a materialized integration approach to avoid data merging each time the data is queried. This decision, however, is a pragmatic decision and the process itself is independent of the chosen storage model.

1.2 Conflicts in Data Integration

There naturally exist numerous differences between data sources and these differences are the main hindrance for developing a general data integration solution. First, data sources are designed and maintained by people with different background and qualification resulting in the same object being modeled in different ways. Second, the systems that are used for data management have different concepts for data modeling and different capabilities regarding the enforcement of integrity constraints. Third, data sources also differ regarding effort devoted towards ensuring data of high quality being stored in these sources. Due to these differences, there are many heterogeneities and quality problems that need to be solved in data integration. These problems are referred to as *conflicts in data integration*. Several classifications of conflicts in data integration exist (see for example [KCGS93, KS91, SPD92]). In general, we distinguish between schema conflicts and data value conflicts.

Schema Conflicts

The schemas of overlapping data sources usually show many differences due to afore mentioned factors that affect schema design. Common conflicts between schemas are:

- Using different sets of attributes to model a real-world object.
- Using different value domains for the same attribute or property.
- Different granularities in modeling an object or object properties. A property that is modeled as an attribute in one schema may be modeled as an individual object (for example a relation) in another schema.
- Homonyms between the schemas. In some cases the same attribute or object name is used to describe semantically different things.

Schema conflicts are solved by schema mappings that describe how data managed under different schemas is to be represented uniformly under the global schema. Schema mappings also resolve heterogeneities in the data models of the sources.

Value Conflicts

Value conflicts occur whenever the same fact or property is represented by duplicate records with different values. These differences are either due to at least one of the representation being erroneous or due to the usage of different value representations. For example, the values '50 °C' and '122 °F' adequately describe the same temperature using different representations. A special kind of conflict is the case where only one of the duplicate records provides a value for an attribute while the other does not (referred to as a NULL VALUE). This special case is called an *uncertainty*.

There exist different strategies to cope with value conflicts in data integration. *Bleiholder and Naumann* give a classification of conflict handling strategies in [BN06]. The three top-level classes of their classification are conflict ignorance, conflict avoidance, and conflict resolution:

- **Conflict Ignorance**: Conflict ignorance describes a strategy that does not make any decision on existing conflicts at all. In a typical implementation, the conflicting values are simply passed on to the user who has then to decide how to handle them. Such a strategy is always applicable and easy to implement. It gives the user the most information about the available data. In case of large numbers of conflicts, however, the user may soon be overwhelmed by the conflicting data.
- **Conflict Avoidance**: The main feature of conflict avoiding strategies is the decision to handle conflicts and pass only one value to the user. The decision on how to resolve conflicts, however, is made before even looking at the values, i.e., conflict handling is specified in advance. Typical implementations of conflict avoiding strategies are to pass only those values to the user that are consistent and return a special value in case of conflicts. For uncertainties one would return only the existing value. Another strategy is to always pass on the value from one particular source. Conflict avoidance strategies are efficient from a computational point of view. They are not efficient from a data quality point of view, since not all available information is taken into account.
- **Conflict Resolution**: In contrast to the previous two classes, conflict resolution strategies do regard all the available information before deciding on how to resolve a conflict. *Bleiholder and Naumann* further divide conflict resolution strategies into deciding and mediating. A deciding strategy chooses the solution from all the present values. A typical example is to always take the

largest or the smallest of the conflicting values. Mediating strategies on the other hand may choose a value that does not necessarily exist among the conflicting values. Instead, these strategies choose a new value or compute a value from the conflicting ones. A typical example is to compute the average of the conflicting values. Conflict resolution has the overall best potential of producing data of high quality, due to the complete consideration of the available information. However, from a computational point of view these strategies are also the most expensive ones.

Our focus is on supporting conflict resolution. We will refer to value conflicts as conflicts in the remainder of this thesis. While conflict resolution is an integral part of data integration, it has received only little research attention so far. In general, conflict resolution is done using resolution functions. A *resolution function* takes two or more values from a certain domain and returns a single value from the same domain [NH02]. Additional values may be used as input for conflict resolution. An overview of conflict resolution functions can be found in [BN05, NH02]. A *conflict resolution strategy* is a collection of conflict resolution functions being defined for individual attributes that are applicable under certain conditions in a specified order.

Deciding on what value is to be taken from a given set of conflicting values or how a solution is to be computed requires input from an expert user familiar with domain constraints, regularities, and possible pitfalls in the data generation process. Due to these dependencies, automatic conflict resolution appears impossible. Instead, individual approaches have to be defined. The algorithms presented in this thesis are intended to assist in defining effective conflict resolution strategies for integrated systems. Effectiveness of conflict resolution is measured by the gain in quality of the final result compared to the original data sources. Insights about reasons for conflicts between given data sources have a major influence on the overall quality of the final result. Only if we have a proper understanding for the causes of conflicts and quality flaws, we are able to eliminate them properly. On the other hand, resolving conflicts is a time-consuming process. Huge amounts of conflicts prevent an expert user from inspecting and resolving each of them individually. Therefore, conflict resolution strategies are normally defined based on conflict samples. After inspecting the sample data, the resulting conflict resolution strategy treats all conflicts alike. The problem with such an approach is the assumption that all conflicts follow the same reason, which not always is true. As a result, the quality of the resulting data set is varying for conflicts following different conflict reasons.

1.3 Problem Statement and Contributions

Depending on the number of conflicts there is a natural trade-off between efficiency of defining a conflict resolution strategy and effectiveness of conflict resolution. In an ideal setting, a conflict resolution strategy is based on independently solving sets of conflicts that share the same conflict reason using a single resolution function for each set. We call such an approach *context-aware conflict resolution*. Context-aware conflict resolution is suitable in situations where differences between data sources are not incidental, but follow some systematic background. A common example for systematic differences is the usage of different vocabularies or measurement units to represent information. Instead of inspecting individual conflicts, the expert user specifies a conflict resolution strategy based on known groups of conflicts that share the same conflict reason. While being more efficient regarding strategy specification, context-aware conflict resolution does not surrender any effectiveness regarding the quality of the final result.

Problem Statement

High quality data is of great importance for scientific research relying on existing databases. Practical experience and quality studies show that databases not always meet the required standards. Data merging has great abilities for cleansing scientific databases due to abundant overlapping data sources. Data merging relies on our ability to identify and solve existing conflicts effectively. Context-aware conflict resolution is an approach to solve groups of conflicts that follow the same systematic reason; a frequent scenario in scientific data sources. Context-aware conflict resolution requires identification of conflicts that originate from the same systematic reason. Information about data generation, however, is rarely provided for existing data sources. Systematic conflicts are therefore discoverable only from the data at hand. Given a pair of contradicting data sources, the problem is to (i) identify conflicts with systematic background, and (ii) give indications towards potential conflict reasons.

Contributions

For the problem statement we see the following contributions:

Classification of Data Cleansing Approaches

Poor quality data has lead to the development of numerous methods for data cleansing. Each method tackles certain aspects of data cleansing. However, a comprehensive overview and classification of these methods does not exist. We give a classification of data deficiencies that diminish the quality of existing data sources and list quality criteria that are affected by these deficiencies. We further provide an overview of existing methods for data cleansing. Based on our classification, we show which cleansing approaches are capable of handling which data deficiencies and quality criteria. We show that data merging has outstanding ability to increase accuracy of data, a quality criteria ignored by most of the existing cleansing approaches.

Data Quality and Data Cleansing in Genome Databases

Through careful analysis of the experimental and annotation process of genome data, we identify different classes for poor data quality. We identify the producers of these errors and pinpoint the employment of each of these producers in the data production pipeline and the types of error they produce. Our analysis provides a sound basis for quality improvement efforts. We show why existing data cleansing techniques fall short for the especially complex domain of genome data. We describe our practical experiences with projects for genome data cleansing that enhance data accuracy by re-annotation and data merging.

Contradiction Patterns – Classification of Conflicts between Overlapping Data Sources

Relying solely on the given information for finding systematic conflicts, identifying meaningful patterns that occur in conjunction with conflicts between contradicting data sources is a valuable indicator for systematic differences. These patterns summarize data properties that are characteristic for conflict occurrence. Conflicts are classified based on these patterns; the patterns act as descriptive information providing insights towards potential conflict reasons. Both, conflict classes and their descriptive information are valuable in support of context-aware conflict resolution. We present a model for systematic conflicts and describe algorithms for efficiently detecting patterns of conflicts in a pair of overlapping data sources. These patterns help in providing answers to questions like “Which are the conflict-causing attributes, values, or value pairs?” and “What kind of dependencies exists between the occurrences of contradictions in different attributes?”.

Minimal Update Sequences – Revealing the process of conflict generation

Contradiction patterns define independent classes of conflicts, but cannot reveal any dependencies regarding the origin of conflicts during data generation. We define a model for dependencies in systematic conflict generation by using sequences of update operations. Sequences of set-oriented update operations are used as abstract descriptions for regular differences among data sources. Given a pair of contradicting databases, each operation may (i) represent an update that has been performed on one of the databases (considering that both have evolved from a common ancestor), or (ii) describe systematic differences in their respective data production processes. Update sequences give valuable insights why a database is different from its original state. Even though we only consider a restricted form of updates, our algorithms for computing minimal update sequences for pairs of databases require exponential space and time. We show that the problem is NP-hard for a restricted set of operations. However, we also present heuristics that lead to convincing results in all examples we considered.

1.4 Outline

This thesis is structured into three main parts. Part I introduces the basic concepts of data quality and data cleansing. We also show that there exist numerous quality problems in existing scientific data sources that are inherently linked to the data usage and production process. Part II introduces our model for systematic conflicts and presents different algorithms for mining contradiction patterns. Part III shows our work on finding minimal sequences of update operations that reflect differences in data generation. We implemented all algorithms using Java™ J2SE 5.0. The experiments where performed on a CITRIX METAFRAME™ Server containing two Intel Xenon 2,4 GHz processors and 4 GB main memory.

Chapter 1 - Introduction

This chapter motivates data cleansing using overlapping data sources. Especially for scientific data, overlapping sources are often the only source of information about hot-spots of poor data quality. We give examples for existing overlapping data sources and argue that integration of these sources indeed helps improve the data quality. We outline the process of merging overlapping sources giving an emphasis on conflict resolution. While there is a rich body of work on data integration not much research effort has been devoted on supporting conflict resolution. We present the problem of discovering descriptions for systematic conflicts that later can be utilized for effective context-aware conflict resolution.

Chapter 2 - Comprehensive Data Cleansing

Common definitions for data cleansing allow a wide range of applications to be considered as cleansing applications. Within this chapter, we give an overview of existing data cleansing approaches based on a classification of data deficiencies and quality criteria affected by these deficiencies. We outline the cyclic process of data cleansing and give a general description of methods used within this process. Our classification of data deficiencies, data quality criteria, and data cleansing methods allows for better comparison and evaluation of existing and future data cleansing approaches. A comparison of existing cleansing methods reveals that the combination of duplicate detection and conflict resolution has great potential for data cleansing. Our review shows that duplicate detection is an integral part of many data cleansing approaches while conflict resolution is often left aside.

Chapter 3 - Quality Issues in Genome Databases

Scientific research today is based in large part on processing and analyzing existing data. Within this chapter, we show that genome data is dirty and that dirty data is caused by inadequacies of the data production process. We give a description of the general production process for genome data and present typical cases of errors from the literature and from our own experiences. Most errors cannot be avoided simply by changing parts of the production process leading to the problem of eliminating them through data cleansing methods. Existing cleansing methods, however, are not applicable for the major errors found in genome data. We list existing approaches for detecting and eliminating erroneous scientific data. We further identify three main challenges for cleansing genome data and discuss two of our practical studies for genome data cleansing, involving re-annotation and merging of overlapping data sources.

Chapter 4 – Mining for Patterns in Contradictory Data

Within this chapter, we present an algorithm for comparing pairs of overlapping databases. The algorithm finds conflicts that occur in some sense systematically or follow certain patterns. These patterns are a special kind of association rules and provide a quick way to find quality hotspots in two data sets. Association rules are a popular concept for knowledge representation. Their simple structure and ease of interpretation makes them a natural choice. We start by introducing the general problem of association rule mining. We then define our special class of patterns, called contradiction patterns. We present the data model and the algorithm for finding contradiction patterns. To cope with the large amount of potential contradiction patterns, we define measures of interestingness for them. In the experimental section of this chapter, we discuss practical parameter values for these measures.

Chapter 5 – Classification of Contradiction Patterns

Within this chapter, we present a modified approach towards highlighting systematic differences. The approach is based on condition-action pairs that represent a natural way of describing differences. Conditions define characteristic data patterns that hold in conjunction with conflicts. Actions describe a value mapping that summarizes the conflicting values. This bipartite way of describing systematic differences not only highlights data characteristics in conjunction with conflicts, but also gives information about the conflicting values themselves. Based on properties of the mappings, we present a hierarchical classification of contradiction patterns.

Chapter 6 – Update Distance of Databases

Contradiction patterns and conflict generators highlight characteristic properties for conflicts between values of individual attributes. In this chapter, we use sequences of set-oriented update operations for finding regularities in contradicting databases. Update sequences allow explanations for the whole set of occurring conflicts and not just for those between values of a particular attribute. Update sequences are also able to outline dependencies that exist between conflicts in different attributes that are not revealed when mining patterns for individual attributes only. We describe algorithms for finding minimal update sequences transforming a given database into another. We derive upper and lower bounds for the length of these sequences and present branch and bound algorithms for calculating minimal update sequences.

Chapter 7 – Heuristics and Problem Variations

Computing minimal update sequences is both, computationally expensive and memory consuming. We describe problem variations that reduce the number of possible update sequences to cope with these problems. For large databases, however, computing minimal update sequences is still infeasible using the defined variations. We prove that for a very restrictive class of modification operations the problem is already is NP-hard. We present heuristics for finding minimal update sequences for large databases. In our experiments, we show that the accuracy of our heuristics is surprisingly good.

Chapter 8 – Conclusion

We conclude the thesis in this chapter by giving a brief summary of the covered topics and give an outlook into future work and open issues.

Chapter 2

Comprehensive Data Cleansing

Data has become a valuable asset for business enterprises and research institutions. The invention of data mining and data warehousing technologies shifted the application of data from a passive role of simply recording business activities and experimental results towards more active roles in business planning and decision making. *Jack E. Olson* rightfully notes in [Ols03] that for a long period of time, the primary focus of data management technology has been on supporting the efficient storage, querying, and analysis of data while not much has been done about the actual data itself. Data quality technology has lagged behind these other areas until the lack of managing the actual content of data sources began to emerge as a major problem in new data usage scenarios. The cost of poor-quality data has been estimated by some data quality experts as being from 15 to 25% of operating profit for business enterprises [Ols03]. In a survey of 599 companies conducted by PRICEWATERHOUSECOOPERS, an estimate of improper data management is costing global businesses more than $1.4 billion per year in billing, accounting, and inventory snafus alone [Ols03]. Other impacts of poor data quality include customer dissatisfaction, lowered employee job satisfaction, less effective decision making, and a reduced ability to make and execute strategy [Red98].

Without even realizing it, most of us are somehow affected by poor data quality. In his book "Enterprise Knowledge Management – The Data Quality Approach", *David Loshin* lists several examples of "Data Quality Horror Stories" [Los01]. These stories include common examples like delivery of multiple identical letters to the same recipient due to duplicated records in customer databases. While being annoying to the receiving person, multiple letters add to the cost of a marketing campaign. Another example for costly failure due to poor quality data given in [Los01] is the fate of MARS CLIMATE ORBITER that vanished in 1999 because engineers failed to convert English measures of rocket thrusts to Newton, causing the orbiter to smash into planet mars instead of reaching the orbit safe.

Just as business data, scientific data contain quality flaws that can have enormous economic and medical impact on users and customers. For instance, errors in genome data can result in improper target selection for biological experiments or pharmaceutical research. To bring a handful of new drugs to the market, pharmaceutical companies spend billions of dollars in research [Hen02]. Of thousands of promising leads derived from experimental genomic data only a handful reach clinical trials and only a single drug becomes marketable. Obviously, it is of great importance to base these far-reaching decisions on high quality data.

The devastating impact of improper data has lead to numerous quality assurance programs that are concerned with all aspects of data generation, management, and usage within an organization. For data to be considered of high-quality it has to be accessible and processable by the people and applications that use it, it needs to be valid for the intended use, and it must be verifiable by decision makers to gain confidence in the data they rely upon. Quality assurance includes training of data entry staff, educating users, documenting data generating processes, monitoring and profiling existing data sources, manipulating improper data, and reengineering and re-implementing data extraction and processing applications. A major part of quality assurance activities is devoted towards data cleansing, also called data reconciliation or scrubbing. *Data cleansing* is defined as the process of detecting and removing errors and inconsistencies from data with the goal to improve the data quality [RD00]. In [MF03], we give a comprehensive classification of data problems that downgrade the quality of data sources. We define the process of data cleansing and describe methods frequently used within this process. This chapter is a revised and extended version of our previous work. We show how the integration of overlapping data sources fits into the data cleansing process, and discuss open questions regarding assistance for conflict resolution.

2.1 Quality Deficiencies in Databases

Before describing data properties that diminish the quality of data, we need to define terms and concepts to describe data and data sources. We follow a formal model of structured data sources that is oriented towards the concepts and notations of relational databases defined in [Cod70].

2.1.1 Structured Data Sources

Data are known facts that can be recorded and that have an implicit meaning. Data is collected to represent part of the real-world, called the mini-world [EN00]. Within this thesis, we consider structured data sources that are collections of records where each record represents a real-world object or fact. Each record is composed of values that are symbolic representations of object properties. For example, the employees in a company may be represented in a data sources with their Social Security Number (SSN), name, address, and salary as recorded properties.

We assume the existence of a non-empty set of domains $D = \{D_1, \ldots, D_m\}$ whose values are sequences of symbols from a finite, non-empty alphabet Σ_D. For each domain D_i, $1 \leq i \leq m$, there exists an arbitrary grammar describing the syntax of the domain values. Domains represent regular languages, i.e., words from Σ_D^* generated by a grammar $G(D_i)$, called the domain format. A relation schema $R(A_1, \ldots, A_n)$ is composed of a relation name R and a list of attributes. Each attribute $A \in R$ is the name of a role played by some domain from D in relation schema R. The domain of each attribute A is denoted by $dom(A)$, with $dom(A) \in D$. The degree of a relation schema R, denoted by $|R|$, is the number of attributes in R. A relation (or relation instance) r of schema R is a set of tuples $t \in dom(A_1) \times \ldots \times dom(A_n)$. Attribute values of a tuple are denoted by $t[A]$. Each tuple is a list of values, where each value $t[A]$ is an element of $dom(A)$. The degree of a tuple, denoted by $|t|$, is the number of values in the tuple.

A database schema $S = (\{R_1, \ldots, R_k\}, \Gamma)$ contains a set of relational schemas $R_1, \ldots, R_k$ and a set of integrity constraints Γ. As a model of some aspect of the real-world, the data in the database should be a precise abstraction and accurate expression of the objects in the mini-world. Integrity constraints are

used to restrict the set of valid instances for a database or relation schema, i.e., to implement conditions that have to hold for abstract representations of the mini-world. A database $s = \{r_1, ..., r_k\}$ is a set of relations satisfying the constraints in Γ, where each relation r_i, $1 \leq i \leq k$, is an instance of relational schema R_i. Each integrity constraint is a function that associates a Boolean value with a database, indicating whether the database satisfies the constraint or not. Within this thesis, we use the term database to emphasis our focus on structured data sources that follow (or are expected to follow) a given schema.

Example 2-1: A structured database for employees of a fictitious company containing several data deficiencies. The database consists of a single relation having five attributes that lists the name, department, birth date, employment date, and current salary for employees.

Employee

NAME	DEPARTMENT	DATE OF EMPLOYMENT	DATE OF BIRTH	SALARY
Lee, Peter	Sales	10/1/1990	12/8/68	10,000 $
Miller, Tom	Accounting	5/1/1891	9/23/1962	5,000 €
Lee, P.	Field Manager	01.10.1990	08.12.1968	10k $
John Smith	Management	5/15/1982	7/19/1959	22,000 $
John Smith	Management	5/15/1982	7/19/1959	22,000 $
Parker, Tony	Sales	⊥	⊥	11,500 £

■

2.1.2 A Classification of Data Deficiencies

There are several factors that influence the quality of a database. We show in the following that these factors not necessarily need to be errors alone. Missing information and deficiencies in data representation are examples for other factors that diminish data quality. We will refer to these factors as *data deficiencies* in the following. According to [MWO] a system that is deficient is "*lacking in some necessary quality or element*". By using the term data deficiency for quality diminishing factors we emphasis that these deficiencies downgrade the overall quality of a given databases. Data deficiencies may occur on any level of the data hierarchy defined by databases, relations, tuples, and data values. We roughly classify data deficiencies into syntactical, semantic, and coverage deficiencies. *Syntactical deficiencies* describe characteristics concerning the format and values used to represent the mini-world. *Semantic deficiencies* prevent a database from being an exact and non-redundant representation of the mini-world. *Coverage deficiencies* prevent a database from being a comprehensive representation of the mini-world. The following definitions are based on our own experiences and on data problems being listed in literature [LLL01, MM00, Mot89, Ols03, Pyl99, RD00]. We use the database in Example 2-1 to exemplify our descriptions.

Syntactical Deficiencies

Lexical errors name discrepancies between the structure of a data record and the specified format. In databases with lexical errors the number of values for at least one tuple t is unexpected low/high regarding the degree of the anticipated relation schema, i.e., $|t| \neq |R|$. For example, assume the data to be stored in a spreadsheet where each row represents a tuple and each column an attribute. If some rows

contain fewer columns than specified by the relational schema, the actual structure of the data does not conform to the specified format. Data that is lexically correct can be parsed into a specified token structure deducible from the schema. Lexical errors do not occur in relational databases under the control of a relational database management system (RDBMS).

Domain format errors specify situations where a given value for an attribute *A* does not conform to the anticipated domain format *G(dom(A))*. For example, attribute *NAME* in Example 2-1 follows the domain format *Surname, Forename*, i.e., $G(dom(NAME)) = \Sigma_D^*$",."$\Sigma_D^*$. While value 'John Smith' is possibly a correct name it does not satisfy the defined format for attribute values. Another example is the usage of different date formats in attribute *DATE OF BIRTH*. Unfortunately, domain formats cannot be explicitly specified in schema definitions for most data management systems thereby disabling their ability for domain format enforcement.

The data deficiency classes lexical errors and domain format errors are often subsumed by the term *format error* or *syntax error*, because they represent violations of the correct syntax format as specified by a given database schema.

Irregularities describe the non-uniform or unexpected use of values, units and abbreviations. The non-uniform use of units and abbreviations result in equal facts being represented by different values. For example, the values '10,000 $' and '10k $' in attribute *SALARY* in Example 2-1 are assumed to represent the same amount of money, but using different units. Another example is the usage of different currencies in the same attribute. Irregularities may lead to misinterpretations, especially true if units are not explicitly listed but are different from those that are expected (recall the fate of Mars Orbiter mentioned in the introduction of this chapter).

Semantic Deficiencies

Incorrect values are the most apparent and harmful type of data deficiency. An incorrect value is considered an error, i.e., a "*difference between an observed or calculated value and a true value*" [MWO]. Identification of incorrect values depends (i) on our ability to formally describe correct values, or (ii) to gather information about the correct value for a certain object property. The complexity of the real-world and our restricted access to the objects of the mini-world makes identification of incorrect values a challenging task. However, it is also the most important task, since nobody wants to base decisions about business or research investments on erroneous data.

Ambiguities are caused by usage of abbreviations. While abbreviations may lead to irregularities, they also allow a multitude of different interpretations. For example, forename 'P.' in 'Lee, P.' in Example 2-1 may be interpreted as an abbreviation for 'Peter' in case that tuples 1 and 3 are regarded as duplicates. However, the value might as well represent a different forename like 'Paul', 'Paula', 'Patricia', etc. Determining the actual value behind a given ambiguity is one of the critical steps in duplicate detection where a decision on the similarity of tuples has to be made.

Duplicates are different tuples that represent the same object from the mini-world. Duplicates are not necessarily identical in all of their attribute values. These duplicates are called inexact or approximate duplicates. Contradicting values between duplicates form a special class of data deficiencies (see below). These deficiencies, caused for example by abbreviations, are the main hindrance for efficient duplicate detection. In Example 2-1 the records in rows 4 and 5 are exact duplicates while the records in rows 1 and 3 are most likely inexact duplicates.

Invalid tuples are tuples that do not represent existing (or valid) objects in the represented mini-world. If any of the employees represented in Example 2-1 is no longer employed by the company the corresponding record in the database represents an invalid tuple if not deleted. Just like incorrect values, detection of incorrect tuples is extremely complicated due to insufficient rules and information that help indicate invalidity of a tuple.

Coverage Deficiencies

Missing values are the result of omissions while collecting or generating data. There are various representations for missing values in databases. While a missing value might just be represented by an empty value, e.g., an empty text, in other cases missing values are also represented by special character sequences, e.g., a date value of '01-01-1900' is often used as replacement for missing information in systems that do not allow insertion of empty values. Missing values may also be represented by a special NULL value. NULL is usually denoted by the special character $\perp$ in relational databases. In any case, whenever a missing value is encountered one has to decide whether a particular value should have been recorded for the particular object property or not. Only in the former case missing values are considered data deficiencies. Thus, detecting missing values does not only involve decoding their representation, but also deciding whether a value is to be considered as missing or not. In Example 2-1 there are two missing values as every employee is expected to have a birth date and employment date.

Missing tuples result from omissions of complete objects that are existent in the mini-world, but not represented by tuples in the database. Any employee working for our fictitious company who is not represented in Example 2-1 causes a missing tuple in the database. Missing tuples are significantly harder to detect than missing values since there is no corresponding representation for them in the data. In many cases an inventory of the represented mini-world is impractical due to the involved cost or the inability to access all the objects in the mini-world again. Therefore, integrity constraints and overlapping databases are usually the only practicable way for detecting missing tuples.

Special (Sub-) Classes of Data Deficiencies

Integrity constraint violations are caused by semantic or coverage deficiencies. Integrity constraints define conditions that have to hold for valid database states; their violation is mainly caused by either invalid values or invalid tuples. Consider a simple integrity constraint stating that the date of birth for an employee has to be before his/her date of employment in a company. While such a constraint represents a very obvious requirement for a person's employment it is not satisfied by the second tuple in Example 2-1. In general, any Boolean function that takes a database instance as parameter is considered an integrity constraint. There are three special kinds of integrity constraints that are commonly used in relational database systems; Key Constraints, Not Null Constraints, and Foreign Key Constraints. *Key Constraints* ensure that the values in a set of attributes uniquely identify each tuple in a relation. Different tuples with identical keys are considered duplicates. *Not Null Constraints* disallow Null values in an attribute and are violated by them. *Foreign Key Constraints* express relationships between tuples (or objects) and are violated by incorrect values or by missing tuples.

Outdated data is special case of incorrect values or invalid tuples. We expect a database to represent the state of the mini-world at a certain point of time or within a certain time frame. Therefore, data representing objects at a time outside of the anticipated time frame is considered incorrect. For example, if the salary of 'Tom Lee' has recently been raised to '6,000 $', the database in Example 2-1 no longer represents the current status for all the companies employees.

Imprecise data are values that are not sufficiently exact or precise for the representation of a certain object property. Imprecise data may have large impact while being only a small deviation from the actual value, dependent on the application of the data. Therefore, we consider imprecise data as incorrect values from the standpoint of their desired usage. Since we are unable to measure real-world facts with absolute exact precision, it is dependent on the data user to define which values are of sufficient precision and which are not.

Contradictions (**Conflicts**) are a special class of deficiencies that only occur between inexact duplicates. Contradictions or conflicts denote cases where different values are given for the same object property. Contradictions result from irregularities or incorrect values. For example, the duplicate representation of employee *Peter Lee* in Example 2-1 has conflicts in each of the attributes. Thereof, the contradiction in attribute *DEPARTMENT* is due to incorrect values while the other conflicts are due to different values representing the same fact. We use the terms contradiction and conflict synonymously throughout this thesis.

Uncertainty describes the case of duplicate tuples where one tuple possesses a value for an attribute *A* and the other tuple does not, i.e., the pairing of a given value and a missing value. Uncertainties steam from missing values, different representations of NULL, or from incorrect values, i.e., a given value for a non-existent property.

Inconsistencies are either integrity constraint violations or contradictions. In general, the term inconsistency is frequently used to describe problems in data integration. In this thesis, we use the term in a more restricted way. Integrity constraint violations are considered inconsistencies. A given database state is not consistent with our understanding of the measured mini-world whenever it violates the integrity constraints expressing our understanding of regularities in the mini-world. To be more precise, integrity constraint violations are *intra-database inconsistencies* given the fact that the current database state is not consistent. Contradictions are *inter-database inconsistencies*. Contradictions refer to inconsistent representation of the same real-world object in different databases. These inconsistent representations may result from irregularities or from either class of incorrect values, e.g., outdated or imprecise data.

Table 2-1 summarizes the occurrence of data deficiencies at different levels of the data hierarchy. Nearly all deficiencies appear on the value or on the tuple level. Conflicts and uncertainties are of special kind. While being caused by deficiencies at the value level they are solely detectable between duplicate tuples. For relations and databases the only deficiencies are inconsistencies caused by violations of inter-relation or intra-relation integrity constraint. However, these inconsistencies result from deficiencies on the tuple and/or value level in one or all of the participating relations.

Table 2-1: Data deficiencies at different levels of the data hierarchy.

Data Hierarchy	Data Deficiencies
Value	Domain Format Error, Irregularity, Incorrect Value, Ambiguity, Inconsistency, Missing Value, Outdated Data, Imprecision
Tuple	Lexical Error, Duplicates, Invalid Tuple, Inconsistency, Conflict, Uncertainty, Missing Tuple, Outdated Data
Relation	Inconsistency
Database	Inconsistency

2.2 The Quality of Databases

Despite the growing awareness for high-quality data, there still does not exists a clear and commonly accepted definition for what data quality means. The most popular and frequently cited definition is "*fitness for use*" [Jur88, WS96, TB98]. While this definition is very intuitive it is non-operational, because data quality cannot be quantified on the basis of this definition. In the literature it is generally agreed upon that quality is defined in multiple dimensions. Frequently mentioned dimensions are accuracy, completeness, timeliness, believability, and relevance. There exist several different definitions and categorizations of quality criteria in the literature [CZW98, Hin02, JV97, Nau02, Red96, SLW97, Wei99]. Existing definitions are for example driven by business applications and information management [Red96], data warehousing projects [Hin02, JV97], or by querying and integrating heterogeneous data sources [CZW98, Nau02]. Theses definitions overlap significantly. However, there is neither a general agreement on data quality dimensions nor is there a rigorously defined set of data quality dimensions. We adhere to the definition of data quality being a composition of multiple quality criteria and define a set of quality criteria affected by data deficiencies. Our focus is on data cleansing and we completely ignore subjective quality criteria like believability and relevance. Data cleansing may also include structural transformation of data. However, the quality of database schema is not our concern and will therefore not be listed as separate quality criteria below. We follow a database perspective of data quality, where data quality means just the quality of the values and the database instance [ORH05]. For being of high quality by our definition data has to be free of any data deficiencies. For each quality criteria, we describe how to assigning numerical values for this criteria in quality assessment. Quality assessment defines the need for and the success of data cleansing. In the following we assume a database r following schema $S = (\{R\}, I)$. The set of objects in the mini-world is denoted by O. We do not assume enforcement of integrity constraints by the system that manages the data.

Accuracy as defined in [Nau02] is the quotient of the number of correct values and the overall number of values in a database. In our definition, we also include correct tuples, i.e., the quotient of the number of correct tuples and the overall number of tuples in the database. Thus, accuracy is an aggregated value over two quality values that are downgraded by incorrect values and invalid tuples respectively. Intuitively, in a database that is perfectly accurate each tuple and each value is a correct representation for an object or property from the mini-world. Accuracy is considered the most important quality dimension and is often used synonymously with data quality [Nau02, Ols03].

Clearness is the quotient of the number of unambiguous values and the overall number of values in a database. A database that is perfectly clear does not contain any abbreviations or other ambiguities that allow multiple interpretations or substitutes.

Completeness is another aggregated quality criteria. On tuple level, completeness is defined as the quotient of objects from O being represented by a tuple in r and the overall number of objects in O [Mot89]. Completeness on value level is also referred to as density. Density is defined as the quotient of existing, non-NULL values in the tuples of r and the number of total values that ought to be known, i.e., that represent a recordable property of an object represented in r [Nau02]. Non-existent object properties that are represented as NULL do not downgrade data quality. A database that is complete represents each object of the mini-world and lists a non-NULL value for each measurable (or recordable) object property.

Schema conformance is defined on tuple level as the quotient of tuples in *r* that conform to the syntax defined by schema *R* and the overall number of tuples in *r*. On the value level, schema conformance is defined as the quotient of values that conform to the defined domain format and the overall number of values. All tuples and values in a schema compliant database follow the same format and are automatically processable without exceptions. A RDBMS enforces schema conformance on tuple level. However, conformance on value level is usually not enforceable beyond the usage of standard data types like STRING, INTEGER, or DOUBLE.

Uniformity describes the proper use of values within each attribute. Uniformity is the quotient of attributes not containing irregularities in their values and $|R|$, the total number of attributes. In a database that is uniform, equal object properties or facts are represented by only one value for each occurrence.

Uniqueness is the quotient of objects represented by tuples in database *r* and the total number of tuples in *r*. A database that is unique does not contain any duplicates, i.e., the database is free of redundancies, conflicts and uncertainties.

Table 2-2 lists the defined quality criteria and the according data deficiencies that downgrade them on either the tuple or the value level.

Table 2-2: Quality criteria and data deficiencies that affect them on the value and the tuple level.

Quality Criteria	Value Deficiencies	Tuple Deficiencies
Accuracy	Incorrect Value	Invalid Tuple
Clearness	Ambiguities	×
Completeness	Missing Value	Missing Tuple
Schema Conformance	Domain Format Error	Lexical Error
Uniformity	Irregularity	×
Uniqueness	×	Duplicate

2.3 Data Cleansing Workflows

We refer to the entire set of operations performed on existing databases to identify and eliminate data deficiencies as data cleansing. Data cleansing operations perform format adaptation for tuples and values, derive missing values from existing ones, merge duplicate tuples, remove contradictions within or between tuples, and correct invalid data values. The goal of comprehensive data cleansing is to receive a database that does not contain data deficiencies. Such a database is considered an accurate, complete, non-redundant, and consistent representation of the mini-world.

2.3.1 A Process Perspective on Data Cleansing

According to [MM00, RH01], the process of data cleansing comprises the three major steps (i) auditing the data to identify data problems that diminish quality, (ii) choosing appropriate methods to (semi-) automatically detect and eliminate such data deficiencies, and (iii) applying these methods to the database. We add another task, the post-processing or control step to examine the results of the cleansing activities to perform exception handling for entries that could not be corrected by the executed methods. After examining the cleansing results the process is continued by auditing the resulting

database to identify remaining data deficiencies that either were not solved by the executed cleansing operations, or became apparent only after other problems were solved. The cyclic data cleansing process, as outline in Figure 2-1, continues until the data has sufficient quality. Specification of the data cleansing process and control of its execution is guided by domain experts and their knowledge about the regularities and peculiarities of the mini-world. Achieving optimal quality, i.e., eliminating all data deficiencies, is usually impossible. Many data problems are difficult to identify and eliminate. This is especially true concerning correctness and completeness since the mini-world is usually subject to constant change and access to the objects is limited. Depending on the intended data use one has to decide how much effort is spend for data cleansing to assure sufficient data quality.

Auditing the Data

The first step in data cleansing is to audit the data to gain information about data regularities and peculiarities. Comparing these results with common domain knowledge and expected value formats enables identification of data deficiencies. Data deficiencies may also be identified as outliers from value or format regularities. Data is primarily audited using statistical methods. Data profiling reveals information about minimal and maximal values, length of text strings, frequency of values, variance and uniqueness of values within an attribute, or occurrence of null values [Ols03]. Values that are unexpectedly small or large have to be considered as potential deficiencies. Methods for automatic structure extraction from strings identify patterns within the values of string attributes [RH01]. These patterns help define domain grammars that were not explicit specified in the relation schema. Data mining approaches identify patterns within database, like functional dependencies [BB95, HKPT98] or association rules [AIS93]. Association rules that are supported by a large fraction of tuples in a database may represent domain rules and can be employed as integrity constraints. Tuples that do not satisfy theses strong association rules are then regarded as inconsistencies [HH04, MML01]. The results of data auditing enable specification of integrity constraints and domain format grammars.

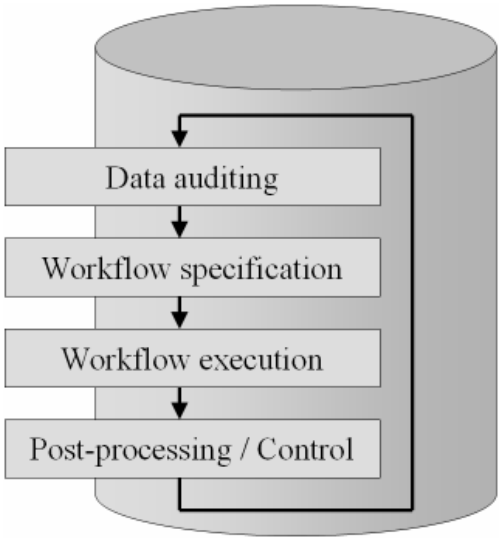


Figure 2-1: The four steps of the cyclic data cleansing process. Steps 1-3 are conform to cleansing processes defined in [MM00, RH01]. The fourth step is motivated by exception handling in cleansing frameworks like AJAX and ARKTOS (see below).

In general, data auditing is performed to identify instances of certain data deficiencies. Auditing may also be performed the find characteristics in the data that occur in data subsets of known inferior quality. Identifying such characteristics helps gaining insights on possible reasons for data deficiencies and can later be used for their elimination. Tools for identification of such regularities are given by statistical methods. Within this thesis we present algorithms that mine regularities in data of inferior quality. Knowledge about reasons and causes for existing data deficiencies is a necessary precondition for the following step in the data cleansing process that chooses appropriate methods for automatic identification and elimination of data deficiencies.

Specifying the Data Cleansing Workflow

Data audition determines the need for data cleansing and highlights hot-spots of poor quality in a given database. Identification and elimination of data deficiencies is then performed in a sequence of cleansing operations. We call this operational sequence the *data cleansing workflow*. Specification of cleansing workflows requires knowledge about the origin of the data. Knowledge about the data generation process and possible pitfalls in that process are the foundation to identify erroneous data and possible reasons for the existence of deficiencies. The causes of data deficiencies are manifold. Typical causes are lazy input habits, inconsistent use of abbreviations, misuse or misinterpretation of data input fields, intentional input of erroneous data, incorrect or careless interpretation of the analysis results, imprecise measurements or systematic errors in experimental setup, or propagation of errors that lead to erroneous analysis results. For the specification of cleansing methods the cause of data problems has to be considered. For example, if we assume that a deficiency results from typing errors at data input, knowledge about keyboard layout can help in specifying and assessing the generation of possible solutions. Within a data cleansing workflow syntax errors are normally handled first. The data is usually automatically processed to detect and remove the other types of deficiencies, a process that is hindered by syntax errors. Otherwise, there is not specific order in handling data deficiencies in a data cleansing workflow. In [RD00] an additional step between specification and execution of a cleansing workflow is defined; the verification step. Within this step, the correctness and effectiveness of the specified workflow is tested and evaluated. We assume this verification to be an integral part of the workflow specification and do not consider it a separate step within the cleansing process.

Executing the Data Cleansing Workflow

The data cleansing workflow is executed after specification and verification of its correctness. The implementation should enable an efficient performance even on large sets of data. There often has to be a trade-off as the execution of many data cleansing operation is computationally expensive, especially if a comprehensive and complete elimination of data deficiencies is desired. Therefore, heuristics are needed that achieve the best accuracy while still having acceptable execution time. An example for a cleansing framework that enables specification and execution of cleansing workflows including heuristics regarding accuracy and performance is the AJAX system (discussed below) [GFSS00, GFS+01]. Usually there exists a great demand for interaction with domain experts during execution of a data cleansing workflow. In several cases the expert may have to decide whether a tuple is erroneous or not and specify or select the correct modification for erroneous tuples or values from a set of possible solutions. Since interaction with the expert is expensive and time consuming tuples that are not corrected immediately are often logged for manual inspection after execution of the cleansing workflow.

Controlling and Exception Handling

After executing a cleansing workflow, the results are inspected to verify the correctness of the specified operations. Within the controlling step data deficiencies that could not be eliminated initially are inspected with the intention to correct them manually. This inspection may result in a new cycle in the data cleansing process, starting by auditing the data and searching for characteristics in exceptional data that allow it to specify an additional workflow to further cleanse the data.

2.4 Methods for Data Cleansing

There exists a multitude of methods and techniques to detect and eliminate data deficiencies. Methods for data cleansing are rarely used standalone. Instead, they are employed in combination with each other within a cleansing workflow. Several methods are even dependent on the results of other methods. For example, conflict resolution depends on duplicate detection in advance, that in turn often uses similarity functions to determine whether tuples are duplicate representations of the same object or not. In the following, we give a short overview for the most common data cleansing methods. Conflict resolution is not further mentioned as a separate data cleansing method. Conflict resolution has already been discussed in Section 1.2 as part of the data merging process. We summarize the data deficiencies detected and eliminated by each of the described methods at the end of this subsection.

Similarity and Distance Measures

Similarity and distance measures are popular examples for cleansing tools that are never used standalone in cleansing workflows, but always in combination with other methods. Similarity is a quantity that reflects the strength of analogy or resemblance between two objects. Distance is a quantity that measures the discrepancy between the two objects. A similarity function assigns a score to a pair of objects, e.g., to data values or tuples. The higher the score, the more similar the objects are. A distance function also assigns a score to a pair of objects. The higher the score, the further apart they are. Distance is regarded as a measure of dissimilarity, i.e., the greater the distance between two objects the lower their similarity. A distance function may be used as a similarity measure and vice versa.

The most popular distance measure for data values is the edit distance of strings. The edit distance between two strings is defined as the minimal number of insertions, deletions, and replacements of characters that are necessary to transform one string into another [Gus97]. The edit distance is for example used in data cleansing to choose a replacement for an incorrect value following the assumption that many incorrect values in databases result from typos and misspellings during data entry. Incorrect values are therefore often replaced by the most similar and correct value, with correctness of values for example being assessed by satisfaction of a given integrity constraint [BFFR05]. *Chaudhuri et al.* present an efficient fuzzy match similarity function for comparing string values that overcomes limitations of the edit-distance [CGGM03]. Their function views strings as sequences of tokens instead of single characters. The varying importance of tokens is recognized by explicitly assigning weight to the tokens. Values matching on high weight tokens are more similar than values matching on low weight tokens. The authors adopt the inverse document frequency weights from IR literature to quantify the notion of token importance. The importance of tokens decrease with their frequency, which is the number of times the token occurs in a reference relation. In Chapter 6, we define an edit-distance for relational instances based on a set of simple set-oriented update operations for databases.

Parsing

Parsing is the process of analyzing and transforming a sequence of characters into a grammatical structure with respect to a given grammar. A parser for a grammar *G* is a program that decides for a given string whether it is an element of the language defined by *G* or not. In the context of compilers for programming languages the parsed strings represent computer programs [AU79]; parsing is performed to detect syntax errors. In the context of data cleansing, parsing is also performed to detect syntax errors. The strings that are parsed are either complete tuples or values coming from a certain attribute domain. When parsing individual attribute values the parser is derivable from corresponding domain format specifications. Parsers that identify tuple structures for data not being managed under the control of a database management system are directly derived from the expected file formats. Tuples or values with syntax errors are deleted, modified, or replaced by correct tuples or values to eliminate syntax errors. Modification is usually performed by data transformations, whereas replacement is typically based on lookup databases (see below).

Data Transformation

Data transformation maps data from a given format into a format expected by the intended application [ACM+99]. Transformation is performed on schema level and on data values. Transformation of data values is called standardization or normalization. These transformations remove irregularities and format errors. Standardization is defined as conversion of values from one domain into another or from one grammatical representation into another. The later usually involves parsing the data first to identify components of the input representation and then rearranging these components into the target representation. A common example is the transformation of persons name values from a representation like <Forename Surname> into a representation <Surname, Forename>. Standardization also includes replacement of abbreviations using an abbreviation dictionary for lookup. Normalization is applied to numeric values. Normalization ensures that all values within an attribute of one or more database relations are within a given interval [SS01]. Normalization and standardization are primarily performed to enhance the comparability and interpretability of a given database.

Schema transformation is usually performed in the context of data integration where data from various sources is mapped onto a global schema. In data cleansing workflows, schema transformation is performed to fit the data into the defined data format. In some cases schema transformation is also performed as a preparation step for duplicate detection, where the data is transformed into a schema that is more appropriate for duplicate detection.

Duplicate Detection

Duplicate detection also called record linkage, entity resolution, or object identification is the most popular technique in data cleansing and has received significant research attention, e.g., [ACG02, BGK+06, HS95, ME97] (see [EIV07, Win99] for surveys). The goal of duplicate detection is to identify multiple representations of the same real-world object. There exist numerous approaches; for relational data as well as for XML data. Duplicate detection for relational data assumes that all tuples follow the same schema. Therefore, data transformations may need to be performed in advance to unify tuple representations. Most duplicate detection methods operate on a single relation. In the case that more than one input relation is given, the tuples are concatenated into a single relation with an additional source identifier attached to them.

Duplicate detection is complex for approximate duplicates. Without a global object identifier duplicate detection requires a function to assess whether different tuples are representations of the same object or not. Quality of detected duplicates is mainly dependent on the quality of the assessment method. Specification of duplicate assessment functions is usually domain dependent and performed by a domain expert. In [ME97] a domain independent duplicate assessment function is described based on the assumption that tuples are made up of alphanumeric characters. Using the SMITH-WATERMAN algorithm [SW81], the edit-distance between tuples is computed to determine whether a pair of tuples is considered as duplicates or not. *Precision* and *recall* have been used extensively to evaluate the quality of duplicate detection algorithms [WNB06]. *Precision* of duplicate detection is defined as the fraction of true duplicates over the set of identified duplicates. *Recall* is defined as the fraction of true duplicates in the set of identified duplicates over all existing duplicates. The goals of different duplicate detection approaches are either on improving the quality of the detected duplicates (effectiveness) or on saving computation time (efficiency).

An exhaustive approach to duplicate detection pair wise compares all tuples in a relation and computes similarity scores for them. The quality of detected duplicates in this approach only depends on the quality of the duplicate assessment function. However, the required effort is quadratic regarding the number of tuples. In [HS95] the authors describe the SORTED NEIGHBORHOOD METHOD (SNM) that reduces the number of tuple pairings and comparisons. Since SNM does not compare all records duplicate detection is approximate to an extend that depends on specific data properties. In SNM, tuples are sorted by a key constructed from one or more attributes in the relation. Sorting intends to bring duplicate tuples closely together based on key similarity. After sorting, only those tuples within a small window that moves over the sorted relation are compared with each other and assessed by the duplicate detection function. Due to the reduced number of tuple comparisons the recall for SNM may be lower than for the exhaustive approach. In order to improve recall, the results of several passes of duplicate detection using different keys are combined by computing the transitive closure of all discovered duplicate tuple-pairs. *Ananthakrishna et al.* propose an approach to avoid the problem of defining appropriate keys for sorting that is inherent to SNM [ACG02]. Their approach, called DELPHI (Duplicate Elimination in the Presence of Hierarchies), relies on hierarchies that are typically associated with dimensional tables in a data warehouse. Hierarchies are expressed in 1:n-relationships between tuples in different relations. Each tuple in the 1-relation is associated with a set of tuples in the n-relation. The degree of overlap between tuples in the 1-relation is a measure of co-occurrence of tuples in their sets of associated tuples. Tuples with significant overlap are considered duplicates. Methods like SNM and DELPHI focus primarily on efficient duplicate matching. Merging the identified duplicates into a single representative is no further discussed. The SERF project at Stanford University develops a generic approach to de-duplication that interleaves invocation of match and merge operations [BGK+06]. Two records are merged as soon as they are identified as duplicates by the match function. The obtained record is added to the dataset and the matching records are deleted immediately. Adding the merged record to the dataset enables identification of duplicates between merged and original records. This approach increases the recall for situations where the original records show insufficient similarity. While the approach may end up comparing each record with each other, by performing merges and deletions as early as possible, it avoids unnecessary future match comparisons.

Database Lookup

Many trusted authorities provide carefully curated databases that are used as lookups in data cleansing. Common examples are address databases provided by postal services. Access to these databases is usually expensive due to the high cost for their maintenance and the effort to eliminate data deficiencies within them. In scientific research, several institutions and organizations provide controlled vocabularies on various topics like chemical compounds, organism taxonomies, or enzyme names. In data cleansing, trusted databases are queried to examine the correctness of a given value, a given combination of values, and for retrieval of replacements for ambiguities, incorrect values, or missing values.

In the first case, a trusted database is regarded as a complete enumeration of an attribute domain. Occurring attribute values in a given database that are not contained in the trusted database are regarded as incorrect. A significant challenge is to effectively clean a value that fails to match exactly with any value in a lookup database. Given an appropriate similarity function, incorrect values can be replaced by the most similar value in the trusted database. In some cases a combination of values is checked for validity using a trusted database. A common example for address data is the combination of postal code and city name. A postal database containing all valid combinations of postal codes and city names is used to identify invalid combinations of postal code and city name in a given database. Database lookup may also be used to fill-in missing values, e.g., fill-in the name of a city given a valid postal code, or to replace abbreviations.

Integrity Constraint Repairs

Constraint enforcement is considered the oldest technique for data cleansing, dating back to the definition of the FELLEGI-HOLT MODEL OF STATISTICAL EDITING in 1976 [FH76]. The FELLEGI-HOLD MODEL is the foundation of statistical data editing, i.e., methods that can be used to edit (clean-up) and impute (fill-in) missing or inconsistent data values [WC02]. Given a set of edits, i.e., rules or constraints on a data instance, *Fellegi and Holt* were the first to define precisely the information needed for correcting a tuple. We will refer to edits as integrity constraints. A tuple is considered correct if it satisfies a given set of constraints. Originally, integrity constraint enforcement describes the problem of ensuring satisfaction of a set of integrity constraints for a given database after modification by inserting, deleting, or updating tuples [MT99]. There are two different approaches for integrity constraint enforcement, namely integrity constraint checking and integrity constraint maintenance. Integrity constraint checking rejects transactions that violate any of the defined integrity constraints. Integrity constraint maintenance is concerned with identifying additional updates, also called repairs, to be executed with a transaction in order to guarantee that the resulting database does not violate any integrity constraint.

In [EBR+01] the authors show that integrity constraint enforcement is applicable for cleansing of databases that violate a given set of integrity constraints, referred to as constraint repair. The basic idea of constraint repair is to automatically identify “low cost” changes that result in a database where all constraints are satisfied. The set of changes is called a repair for the database and the main goal is to find a repair having minimal cost. Cost is normally stated in terms of tuple insertions and deletions. The authors in [EBR+01] also outline limitations of the application of constraint repair for data cleansing. The main limitation is given by the fact that constraint repair operates on databases that show multiple constraint violations while integrity enforcement operates only on valid database states vio-

lated by a single transaction. For database showing numerous constraint violations the resulting search space of possible repairs becomes large. For instance, the order of repair has to be considered. We could first repair tuple t_1 and then based upon the result repair tuple t_2 or the other way around, leading to different results. In addition, the generated repairs are primarily insertion or deletion of tuples rather than modifications of incorrect values. These repairs tend to insert invalid tuples or delete tuples that are at least partially correct, thereby decreasing the accuracy and/or completeness of the database. Recently, *Bohannon et al.* presented an algorithm for finding repairs based on modifications of attribute values rather than insertion and deletion of tuples [BFFR05]. The cost for database repairs in their setting is calculated using a similarity metric, like the edit-distance, between the original value and the repaired value. The authors prove that finding minimal-cost repairs in this setting is NP-complete and present heuristics that efficiently find repairs with the tradeoff that the cost is not necessarily minimal.

Data Mining and Statistical Methods

Statistical methods have become another popular tool in data cleansing workflows for detecting and replacing incorrect or missing values. Statistical methods are especially helpful in the absences of appropriate lookup databases and comprehensive sets of integrity constraints. Furthermore, statistical methods are used for data analysis in the auditing step of data cleansing. In [RD00] the authors differ between two related approaches for data analysis, data profiling and data mining. Data profiling focuses on the analysis of individual attributes in a relational instance. It derives information such as the length, value range, discrete values, value frequencies, variance, uniqueness, occurrence of null values, etc., that assists an expert user in assessing various quality aspects of the attribute. Data mining helps discover specific data patterns in large data sets, e.g., relationships holding between several attributes. Data mining methods include clustering, summarization, association discovery and sequence discovery [Fay98]. Analyzing data using data profiling and data mining techniques helps to identify 'unexpected' values that are indicating possible incorrect values. Identification of outliers, i.e., data values that are very different from the rest of the data by some similarity or distance measure, is done using clustering or association rule mining algorithms. Outliers are represented by violations to strong association rules or by data points forming unexpectedly small clusters. In many cases outliers are considered incorrect values since they do not adhere to the standard characteristics of the dataset. Association rules may also be used to derive integrity constraints [HH04, MM00, SHM+99]. Initially proposed for data warehouse design, mining strong association rules between attributes is used to induce integrity constraints. Tuples that fail these association rules are considered outliers (or values of poor data quality). Within this thesis, we adopt existing association rule mining algorithms to identify characteristics in contradictory data. Association rule mining algorithms are discussed in more detail in Section 4.1.

Statistical methods may also be used to find replacements for incorrect or missing values. Possible solutions include calculating an average value for a set of contradicting values or regression analysis, i.e., predicting the value of an attribute based on the values of several other attributes (predictors).

Summarization of Data Cleansing Methods

In accordance with our classification of data deficiencies, we refer to the detection and elimination of syntactic deficiencies as syntactic data cleansing and to the detection and elimination of semantic deficiencies as semantic data cleansing. Elimination of missing values is called imputation in statistics [LR87] and we adopt this term for the according cleansing methods. In addition, we use the term augmentation to refer to methods that insert missing tuples in cleansing processes.

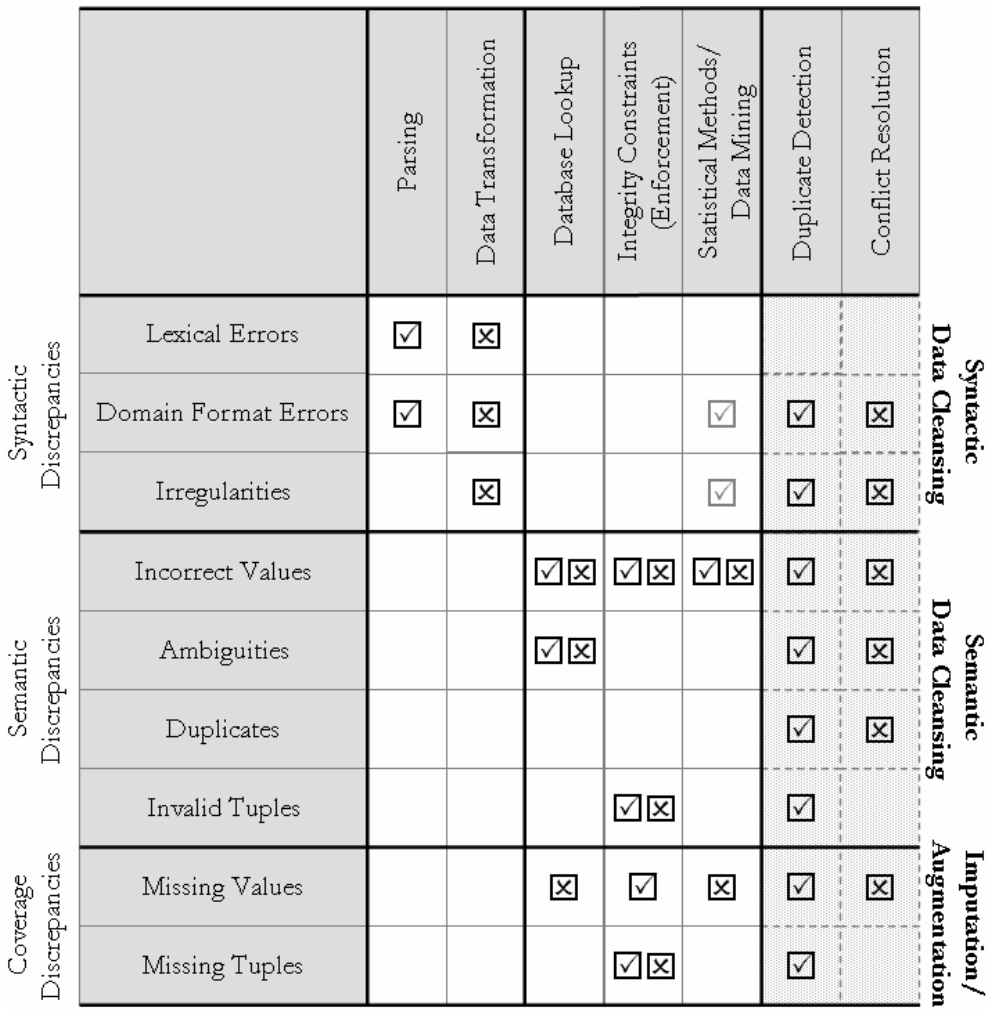


Figure 2-2: Summarization of data cleansing methods that enable deficiencies detection ☑ and/or deficiency elimination ☒.

Syntactic data cleansing is concerned with generating a database that (i) conforms to a given schema and format definition, and (ii) is uniform in its representation of the mini-world. Figure 2-2 shows that parsing, data transformation and duplicate detection coupled with conflict resolution are the primary methods for syntactic data cleansing. Parsing is used to detect syntax errors. Elimination of syntax errors is performed by either deleting the erroneous data or by transforming the data into a format suitable to the defined schema and domain format. Domain format errors may also be detected as inconsistencies between duplicate tuples and eliminated by conflict resolution. Irregularities are eliminated by standardization. Detection of irregularities is far more complicated that detection of syntax errors. In some cases data mining methods may be applicable for the identification of irregularities. Irregular values appear as outliers if they are significantly different from the other values within an attribute. Otherwise, irregularities are primarily identified as conflicts between duplicates.

Semantic data cleansing affects quality criteria accuracy, clearness, and uniqueness. Since accuracy is considered the most important quality dimension, semantic data cleansing has to be considered the most important task in comprehensive data cleansing. However, it is also the most challenging one. The challenge arises from the problem that limited access to objects of the mini-world disables a valid verification or re-measure to ensure correctness of values and tuples. Therefore, we rely on methods that enable us to gain further evidence for the accuracy of a certain value. Semantic data cleansing and

augmentation is supported by database lookup, integrity constraint repair, statistical methods, and duplicate detection coupled with conflict resolution. Database lookup is commonly used to eliminate incorrect values and resolve ambiguities. In some rare cases database lookup also bears the ability to detect and eliminate syntactic data deficiencies. However, the lookup approach requires trusted databases that represent a complete list of values in an attribute domain. Such a comprehensive list is only reasonable for few specific attribute domains, e.g., the names of cities in a country or the names of known chemical elements (periodic table). Constraint violations are the result of invalid or missing data. Therefore, constraint repair helps eliminate these deficiencies. The main limitation of constraint repair is that there normally exist many possible repairs. Repairs are chosen based on a cost function without any further arguments for or against the correctness of values. Another problem is the (in-) ability to specify an adequate set of integrity constraints in many scenarios. Especially in scientific research, knowledge about domain regularities and rules is incomplete and exceptions are numerous thus complicating constraint specification. Data mining methods are used to detect outliers and derive missing values or replacements for incorrect or missing values in semantic data cleansing and imputation. Similar to limitations of constraint enforcement, the accuracy of statistical methods for data cleansing is dependent on the quality of the applied statistical model.

The combination of duplicate detection and conflict resolution (shaded grey in Figure 2-2) shows great potential in all areas of comprehensive data cleansing, especially in semantic data cleansing. First of all, duplicate detection is used to ensure uniqueness for the tuples in a given database. After duplicates are detected conflicts and uncertainties indicate incorrect values, irregularities, and missing values. Domain format errors and ambiguities are also detectable as irregularities whenever duplicates possess different values resulting from different domain formats or use of abbreviations. Data deficiencies are eliminated in a subsequent conflict resolution step using appropriate conflict resolution functions. In addition to detecting value deficiencies, duplicate detection has the potential to detect missing or invalid tuples. Given a pair of databases, missing or invalid tuples are given as tuples in one of the databases without a matching duplicate in the other one. Handling these tuples is specified as conflict resolution on tuple level. Note that data merging also has its limitations. First of all, accurate duplicate detection in itself is a difficult problem. Second, not every incorrect value, irregularity, or missing value within a database is identifiable after duplicates have been detection. Duplicates may contain equal incorrect values or missing values for the same attribute. These situations become more likely the more databases are generated by replication and integration of existing databases.

2.5 Conflict Resolution in Data Cleansing Solutions

We showed the potential of duplicate detection and conflict resolution (or data merging in general) for semantic data cleansing. In the following, we provide a brief overview of existing data cleansing approaches and discuss their support for duplicate detection and conflict resolution. These approaches are often referred to as data cleansing frameworks, since they provide a set of cleansing methods and allow specification and execution data cleansing workflows for arbitrary data sources. Recently, languages have been specified that allow declarative specification of conflict resolution strategies independently of any data cleansing framework.

2.5.1 Existing Data Cleansing Frameworks

AJAX [GFSS00, GFS+01] is an extensible framework for data cleansing wherein the logic of a data cleansing workflow is modeled as a directed graph of data transformations that start from some input data source and result in a clean output data source. AJAXs approach is to take advantage of the functionalities provided by existing DBMSs. AJAX provides a declarative and extensible language for data cleansing based on five logical operators. These operators extend the data transformations expressible in SQL. The operators allow specification of schema transformations, integrity constraint checking, identification of duplicate tuples, and merging of duplicates. The semantic of the operators includes generation of exceptions that provide the foundation for explicit user interaction and the stepwise refinement of cleansing workflows. Tuples that generate exceptions during execution of the cleansing workflow are logged in an exception file and can afterwards be re-integrated into the modified cleansing workflow. The AJAX framework is extensible (i) in that transformations can invoke external domain specific functions, (ii) transformations can be combined with regular SQL statements, and (iii) the set of algorithms implementing the operators can be extended as needed. Recently, the set of operators in AJAX has been extended by a CLASSIFY operator in BIO-AJAX, an extensible framework for biological data cleaning [HGP+04]. Classification is a routine procedure for biological data, especially for annotation purposes (see Chapter 3). Therefore, the CLASSIFY operation is essential in cleansing approaches for this type of data. AJAX separates the logical and physical level of data cleansing workflows. The logical level supports design of data cleansing workflows. The physical level of AJAX executes data cleansing workflows. At the physical level, certain decisions can be made to speed up the execution of data cleansing workflows by selecting among different algorithms that implement logical operators and externally defined functions.

AJAX defines a MATCH operator that is used to identify duplicate tuples based on similarity of their attribute values. Different algorithms that implement the MATCH operator may be provided by the user. Duplicates are clustered in groups and merged using the MERGE operator. Conflict resolution using the MERGE operator requires provision of user-defined conflict resolution functions since AJAX itself does not specify conflict resolution functions other than those provided by the RDBMS used.

POTTER'S WHEEL [RH01] is an interactive cleansing framework that integrates transformation of data with deficiency detection in a single interface. The main motivation for POTTER'S WHEEL is the lack of interactivity of existing cleansing frameworks where transformation is typically done as a batch process. Batch processing leads to long delays and the user has no idea, whether a transformation is effective or not. POTTER'S WHEEL provides a set of operations, called transforms that support common data transformations without explicit programming. Supported transforms are (i) Value translations, that apply a function to every single value in a column, (ii) One-to-one transforms, that are column operations that transform individual rows, for example to unify data collected from different sources, and (iii) Many-to-Many transforms, that help tackle schematic heterogeneities in cases where information is stored partly in data values, and partly in the schema. Users gradually build transformations in POTTER'S WHEEL by composing and debugging transforms, one step at a time, on a spreadsheet-like interface. The effect of transforms is shown immediately for records visible on screen. Deficiency detection is done automatically in the background on the latest transformed view of the data. Data deficiencies are flagged as they are found. The immediate feedback enables the users to gradually

develop and refine the process as further deficiencies are found thus enabling individual reaction on exceptions.

POTTER'S WHEEL does not explicitly mention operations for duplicate detection and conflict resolution. The data deficiencies handled by POTTER'S WHEEL are primarily syntax errors and irregularities. POTTER'S WHEEL allows users to define custom domains and algorithms to enforce domain format constraints. These arbitrary domains are considered domain formats as described in Section 2.1.1. The user does not have to specify the format explicitly in advance. POTTER'S WHEEL lets users specify the desired results on example values and automatically infers regular expressions describing the domain format. The inferred domain format specification can afterwards be used to detect deficiencies.

ARKTOS [VVS+01] is a framework capable of modeling and executing the Extraction-Transformation-Load process (ETL-process) for data warehouse creation. The ETL-process consists of single activities that extract relevant data from the sources, transform it to the target format, and load it into a data warehouse. Therefore, each activity within the process is linked to input and output relations. The logic performed by an activity is declaratively described by a SQL-statement. Data cleansing is considered an integral part of the ETL-process. Each activity is associated with a particular error type and a policy that specifies the behavior (the action to be performed) in case of error occurrence. Six types of errors can be considered within an ETL process specified and executed in the ARKTOS framework. PRIMARY KEY VIOLATION, UNIQUENESS VIOLATION, and REFERENCE VIOLATION are special cases of integrity constraint violations. The error type NULL EXISTENCE is concerned with the elimination of missing values. The remaining error types are DOMAIN MISMATCH and FORMAT MISMATCH, both referring to lexical and domain format errors. The policies for error correction simply are IGNORE, DELETE, WRITE TO FILE, and INSERT TO TABLE. The latter two provide the only possibility for interaction with the user. Similar to POTTER'S WHEEL ARKTOS does not explicitly consider duplicate identification and conflict resolution.

INTELLICLEAN [LLL00, LLL01] is a rule based approach where the main focus is on duplicate elimination. The framework consists of three stages. In the Pre-Processing stage syntactical errors are eliminated and values are standardized including elimination of abbreviations. In the Processing stage cleansing rules are evaluated on data records. These rules specify cleansing activities that are executed on records that satisfy a given condition. There are four different classes of rules. Duplicate identification rules specify the conditions that classify tuples as duplicates. Merge/Purge rules specify handling of duplicates. However, support for conflict resolution in merge/purge rules is not explicitly mentioned in [LLL00, LLL01]. If no merge/purge rule is specified, duplicates are manually merged at the next stage. Update rules specify updates for tuples that satisfy a given condition. Update rules are used for constraint repair and to impute missing values. Alert rules specify conditions under which the user is notified to allow further actions. During the first two stages the actions taken are logged providing documentation of the performed operations. In the verification and validation stage these logs are investigated to verify and possibly correct the performed actions.

HUMMER [BBB+05] is a tool that allows users to specify ad-hoc, declarative merging of data from heterogeneous data sources that contain duplicates, conflicts, and missing values using a simple extension to SQL. HUMMER combines schema matching, duplicate detection, and conflict resolution between heterogeneous data sources to a one-stop solution. HUMMER proceeds in three fully automated steps: (i) instance-based schema matching to bridge the semantic heterogeneity between relation

schemas by aligning corresponding attributes, (ii) duplicate detection, and (iii) conflict resolution by merging duplicates into a single and clean representation. HUMMER optionally visualizes each intermediate step and allows users to interfere. Conflict resolution in HUMMER is performed using the FUSE BY statement that groups the identified duplicates based on a global identifier and allows specification of conflict resolution functions for each of the attributes. The FUSE BY statement is described in more detail below.

2.5.2 Approaches for Declarative Conflict Resolution

RDBMSs allow declarative expression of conflict resolution strategies using standard concepts of grouping and aggregating provided by SQL. Assuming that duplicates are identifiable using a global unique object identifier, tuples are grouped on this identifier to form clusters of duplicates. Duplicates within each cluster can then be merged by resolving conflicts using standard aggregation function like *min()*, *max()*, or *avg()*. Usage of SQL-queries for conflict resolution is limited to the set of conflict resolution functions available in SQL. Most modern RDBMSs allow extensions of the set of resolution function using the concept of user defined function. Over the last years advanced language extensions for declarative conflict resolution have been developed.

FRAQL [SCS00, SS01] is a declarative language that supports data cleansing. The language is an extension to SQL based on an object-relational data model. FRAQL supports the specification of schema transformations as well as standardization and normalization of values using user-defined functions. FRAQL also provides approximate join and union operators. These operators allow detection of duplicate tuples based on similarity of values within one or more attributes. By allowing user-defined reconciliation functions, FRAQL allows merging of identified duplicates within approximate join and union operators. Further cleansing operations supported by FRAQL are filling in of missing values, and eliminating invalid tuples by outlier detection using statistical methods.

FUSE BY: In [BN05] the FUSE BY statement is proposed to overcome limitations of the simple SQL-based approach for declarative conflict resolution. The FUSE BY statement is an extension of SQL that allows specification of powerful and flexible conflict resolution function within the SELECT clause. Examples of conflict resolution functions usable in FUSE BY statements are *choose*(), which returns the value supplied by a specific source, *coalesce*(), which takes the first non-NULL value appearing, and *vote*(), which returns the value that appears most often among the presented value. The WHERE-clause of the FUSE BY statement also allows specification of resolution strategies for a subset of the tuples in the merged relation(s). This ability is of particular interest in our case of context-aware conflict resolution.

2.6 Summary and Related Work

The blurred definition of data cleansing as the process of detecting and removing data deficiencies for quality improvement allows a wide range of application to be considered as data cleansing methods. Furthermore, since data quality is not a well defined concept in itself almost any activity that changes any aspect of a given data set may be considered as data cleansing. For instance, considering data quality as “fitness for use” declares any transformation of data into a format more convenient to an application a cleansing activity. Within this chapter, we take a data centric approach to classify existing data cleansing approaches. Data cleansing is defined as a cyclic process of four steps. Each cycle

executes a set of data cleansing methods to eliminate deficiencies and enhance the quality of a database in the according criteria. We believe that our classification of data deficiencies, data quality criteria, and data cleansing methods together with the outlined relationships between them allows for better comparison and evaluation of existing and future data cleansing approaches.

There exist various other descriptions and classifications of data cleansing problems in the literature [KCH+03, ORH05, RD00]. In [RD00] the authors differ between schema-related and instance-related problems. Schema-level problems can be addressed at the schema level by schema evolution, schema translation, and schema integration. Examples for schema-level problems are missing integrity constraints or domain format errors. Schema-level problems are always reflected by deficiencies within the databases instances. In our definition the cleansing of data is performed on the database instances and does not affect the database schema. We therefore do not distinguish between schema and instance related problems. Data cleansing problems are further divided into single-source and multi-source problems. According to the definition in [RD00] integration of overlapping databases reveals and solves multi-source problems. However, since all multi-source problems on instance level are also present in single-sources we do not distinguish between single-source and multi-source problems. Kim et al. present a detailed taxonomy of dirty data having 33 primitive types [KCH+03]. The taxonomy is oriented towards the causes of dirty data. For example, the authors define a dirty data type named *Non-enforcement of automatically enforceable integrity constraints* including subtypes of dirty data related to the lack of concurrency control, i.e., *Lost Update*, *Dirty read*, and *Unrepeatable Read*. In contrast, our classification of data deficiencies is independent of any error causes since many different causes may result in the same type of data deficiency. Recently, *Oliveria et al.* published a formal definition of data quality problems [ORH05]. Quality problems are defined at different levels of the data hierarchy, i.e., database, relation, tuple, and value. This definition of quality problems is also more detailed than our classification. For example, the authors define separate types of integrity constraint violation, i.e., one for each different type of integrity constraint. Regarding the discussion of cleansing methods our coarse-grained definition of deficiencies is sufficiently detailed.

A comparison of existing data cleansing methods regarding the quality criteria they affect reveals that the combination of duplicate detection and conflict resolution has great potential for semantic data cleansing. At the end of this chapter, we reviewed existing data cleansing approaches and discussed their support for duplicate detection and conflict resolution. The discussion shows that duplicate detection is an integral part of many cleansing approaches while conflict resolution is often left aside. Recently, declarative languages for conflict resolution are defined. These approaches assume that an expert user is capable of defining a conflict resolution strategy. However, none of the approaches mentions support or assistance for the user in specifying a conflict resolution strategy by outlining possible conflict reasons.

While data merging helps increasing data accuracy there are also limitations to this approach. First of all, the approach relies on duplicate detection methods that are capable of identifying duplicates with high precision and recall. Second, matching values are considered as correct values. This assumption is not always true. For instance, *S. Brenner* states in his comparison of functional annotations that cases where different groups arrive at consistent but wrong conclusions are like since all rely on similar data and methods [Bre99]. A last limitation is due to the fact that we will not always be able to derive a correct value from conflicting ones. While the given values point towards areas of poor quality, none of them is necessarily correct and thereby none of them provides an argument towards the correct con-

flict resolution. Thus, the best overall quality for a cleansing approach is given by combining several different cleansing methods. The available resources and the targeted data quality criteria and result define to what extend different methods are used in a data cleansing workflow.

Chapter 3

Quality Issues in Genome Databases

High costs and loss of reputation caused by data of poor quality made quality assurance and data cleansing hot topics in the business world over the past decades. Recently, data quality has become an issue in scientific research as well. Similar to decision making in business enterprises, scientific research is based in large part on processing and analysis of existing data collections. The huge amount of scientific data produced on a daily basis makes manual control of data quality infeasible. Almost all major scientific data repository exhibits an exponential growth in the number of data entries over the past decade (see for example GENBANK statistics [GENB]). Increasing automation in data production and analysis makes scientific data repositories further vulnerable to quality flaws that remain unnoticed by human inspectors. In the following, we use genome data as an example to outline common quality problems in scientific data and discuss approaches for genome data cleansing. We believe that the data problems outlined in this chapter, although being limited to a particular domain, are not restricted to genome research, but are inherent to many other areas of scientific data production, processing, and analysis.

Genome data includes the actual sequences of bio-molecules, i.e., DNA, RNA, and protein, as they were observed in wet lab experiments. In addition to the raw data, genomic databases store structural and functional classifications for sequences and subsequences, called annotations. Annotation data represents the most important part of genomic databases, namely the deeper biological meaning of the raw data. Through careful analysis of the experimental and annotation process of genome data, we identify five classes for poor data quality: experimental errors, analysis errors, transformation errors, propagated errors, and stale data. To tackle the problem of data errors of the five kinds, a first step is to identify the producers of these errors. In [MNF03], we develop detailed Information Product Maps (IP-Maps, [SWZ00]) for genome data production. In general, there are four classes of data- (and thus error-) producers: Wet-lab experiments, semi-automated experiments, computational transformations, and computational analysis. Our analysis pinpoints the employment of each of these producers in the data production pipeline and the types of error they produce, thus providing a sound basis for quality improvement efforts. We show why existing data cleansing techniques fall short for the especially complex domain of genome data. At the end of this chapter, we describe our own experiences in data cleansing and in building a data warehouse of protein structure data from overlapping data sources.

3.1 Basic Concepts for Describing Genome Data

Similar to the term "gene" itself, "genome data" is a term without a clear marked-off scope or commonly accepted definition. The *genome* is the entirety of genetic information of an organism. Genetic information is used by organisms to transform energy from the environment, to reproduce, to self-assemble (grow), and to repair themselves. Genetic information is stored as the sequence of the four different building blocks, called *bases* (*adenine*, *guanine*, *cytosine*, and *thymine*), of the molecule *desoxy ribonucleic acid* (DNA). The DNA – a double stranded molecule forming the well-known double helix – is divided into transcribed and non-transcribed parts. The former are called *genes* and are the parts of main interest in biological, medical, and pharmaceutical research. *Transcription* is the first step of genome information processing. The resulting molecule, *ribonucleic acid* (RNA), is the single strand copy of a gene. It is used as a template for protein synthesis. The synthesis process, called *translation,* uses an organism-specific translation table (*genetic code*) to translate successive segments of length 3 (*codon*) into amino acids that are the building blocks of the resulting protein. The translation always starts at a *start-codon* `atg` and ends at the first *stop-codon*, i.e., `taa`, `tag`, `tga`. The codon-structure defined by the start- and stop-codons is called the *reading frame*. Proteins are the building blocks of living organisms performing a multitude of different functions. This process of biological information processing within an organism's cells is called the "central dogma of molecular biology" as described by Francis Crick in 1957, and is shown in Figure 3-1.

In principle, every piece of information about the genome and genome products of living organisms can be termed genome data. By *genome data* we mean information about the bio-molecules DNA, RNA, and protein, such as their sequence (composition of bases or amino acids), their structural features, and their function performed within the organism. Here, we disregard data from gene expression studies, information about protein interactions during complex biological functions as well as the 3D-structure of molecules (see Section 3.4.3 for an example of quality problems in protein structure databases). The main data for data quality studies in this chapter are:

- **Strings** representing the sequences of bio-molecules,
- **Attributes**, describing certain properties using values from a fixed set of domains, and
- **Annotations**, i.e., functional or structural classification for regions of the genome or proteins.

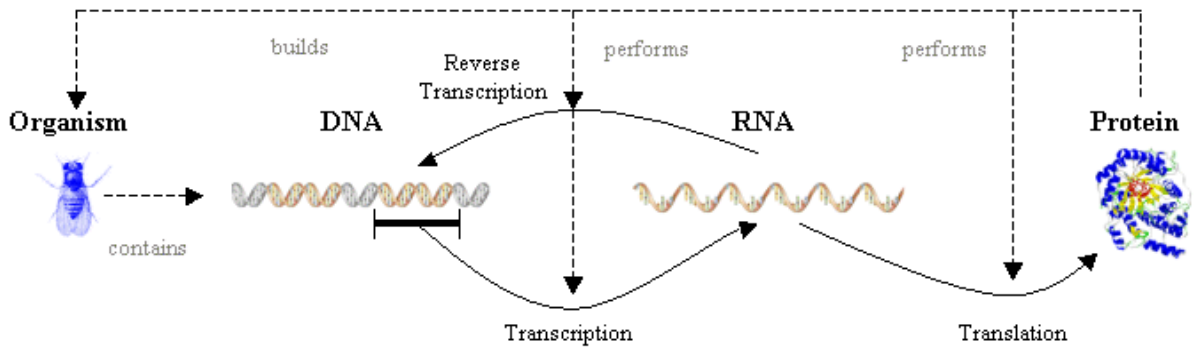


Figure 3-1: The Central Dogma of Molecular Biology defines the process of transcribing DNA molecules into RNA molecules and translating these molecules into proteins. The whole process is performed by existing protein molecules that also act as the main building blocks of the organism.

The data is further classified into semantic classes of genome data resulting from the following general process:

- **Genome sequence data** represents the sequence of DNA molecules extracted from the cells of different organisms. They are represented by strings over the four-letter alphabet {a, c, g, t}. Each string represents either sub-parts of the complete genome or concatenated larger parts.
- **EST sequence data** are also strings over a four letter alphabet representing transcribed parts (RNA) of the genome, called expressed sequence tags (ESTs).
- **Structural annotation** describes known features that are identified and shown on the genome sequence data. The features of interest are for example the occurrence of sequence patterns, *single nucleotide polymorphisms* (SNPs), e.g., proven sequence variation between individuals, and gene location and gene structure.
- **Protein sequence data** represents the sequence of amino acids of proteins by a string over the alphabet of twenty amino acids.
- **Functional annotation** describes in non-standardized textual form the function performed by a certain protein within the organism, as well as its participation (or that of its mutations) in the development of a certain disease. Biologists enter free text descriptions at will, in different languages, using different abbreviations, etc.
- **Protein motifs** represent the conserved characteristic features of a protein family, i.e., groups of related proteins within different organisms in various forms. Often, only small but highly conserved parts of the protein are responsible for a certain function, and within these parts several combinations of amino acids are allowed.

3.2 Genome Data Production

Genome data production is performed by people with different skills and domain knowledge. The process involves:

- **Biologists** working in the wet-lab,
- **Lab assistants** who install and operate machines and robots, and
- **Bioinformaticians**, i.e., computer users having biological expert knowledge.

Production of genome data is done in collaboration by different workgroups and different institutions from around the world, using their own, often proprietary, techniques, methods, and protocols. This setup alone implies the poor data quality of the end product, as we will argue later. The main data-producing techniques for genome data are:

- **Wet-lab experiments** (performed by biologists): Within the wet-lab biologists perform experiments on the living organism and explore interesting features, such as organism behavior after manipulation under given conditions. The results are interpreted and transformed into information stored in digital format within usually proprietary data storage system.
- **Semi-automated experiments** (performed by lab assistants): Automata and robots support biologists in performing those experiments that are dreary and must be repeated often. This automation increases the throughput and lowers the error rate, because machines are able to perform without

fatigue. For example, sequence determination is a very error prone process when performed by humans. Biologists still have to perform the experimental set-up, but the automata generate the information directly in digital format.

- **Computational transformation** (performed by bioinformaticians or lab assistants): Transforms data from one representation into another that is better manageable and interpretable by humans or machines. Transformation involves translation of sequence information (e.g., *base calling*) or concatenation of strings (e.g., *sequence assembly*). Computational transformations do not require additional knowledge-based interpretation of the results.
- **Computational analysis** (performed by biologists or bioinformaticians): The results of experiments are interpreted by human experts using computer software to produce new information. Interpretation of experimental results is termed data analysis in general, and plays an important role in genome data production. Here, digital information is interpreted to generate new digital information. The role is important due of the huge amount of data produced, mainly by semi-automated experiments, that is to be analyzed. Often, a genome data product can be derived alternatively by wet-lab experiments or computational analysis. There is a trade-off in terms of quality involved, as experiments are more accurate than computational analysis, while also being more expensive and time consuming.

From our description it already becomes clear that genome data production is an interdependent process. The information gained in one step is used and further analyzed in the following step, generating new knowledge and information. The information gained is eventually re-used as input in further data generation and analysis. The overall process is shown in Figure 3-2. Genome data is produced in four (mostly) dependent steps:

Step 1 **DNA sequence determination**: Starting from the living organism, the sequences of the DNA (genome sequence data) and of the transcribed genome regions (EST sequence data) are generated. DNA sequence determination is performed in a combination of wet-lab experiments, and semi-automated experiments, and also includes computational transformation. The results are strings representing DNA sequences and attributes describing the sequence properties, such as the organism it was taken from.

Step 2 **Genome feature annotation**: After DNA sequence determination relevant biological regions and structural features are identified on the genome sequence data, e.g., the localization and structure of genes. In general, the process of assigning meaning to sequence data by identifying regions of interest and determining biological function for those regions is defined as genome annotation [FGM+03]. The data is mostly generated by computational analysis or it reflects the results of other experiments mapped onto the genome sequence using computer programs. This step produces structural classifications of sequence regions.

Step 3 **Protein sequence determination**: The sequence of proteins is determined either experimentally using extracted proteins from the living organism or by computational transformation of information from the previous two steps. In the first case, the production mainly is performed experimentally, because semi-automation is only marginal within this process. In the second case, determination of protein sequences is performed by simple translation of the genome sequence of the identified genes. In both cases, the result is a string representing the amino acid sequence of a protein.

Step 4 **Protein function annotation**: Using the protein sequences resulting from protein sequence determination, the function of the protein performed within the organism is described. Functional annotation of proteins is normally done by assigning the protein to different classes of biological function based on features of the amino acid sequence. Protein functional annotation can be performed either experimentally, which is time consuming, or computationally, which is fast but error prone. The result is a set of functional classifications for each protein.

In recent years, a multitude of tools and protocols for producing genome data have been developed (see for example [AFV94, BO04]). The usage and combination of these tools and protocols within the genome data production process varies among institutions and workgroups and also changes over time. In most cases these changes remain undocumented for the outside world - making it hard to reconstruct the production process. A standard procedure for genome data production does not exist. Therefore, we give only a general overview of the basic steps involved. The occurring errors that influence the quality of the resulting data in the different sub-processes are described in the following.

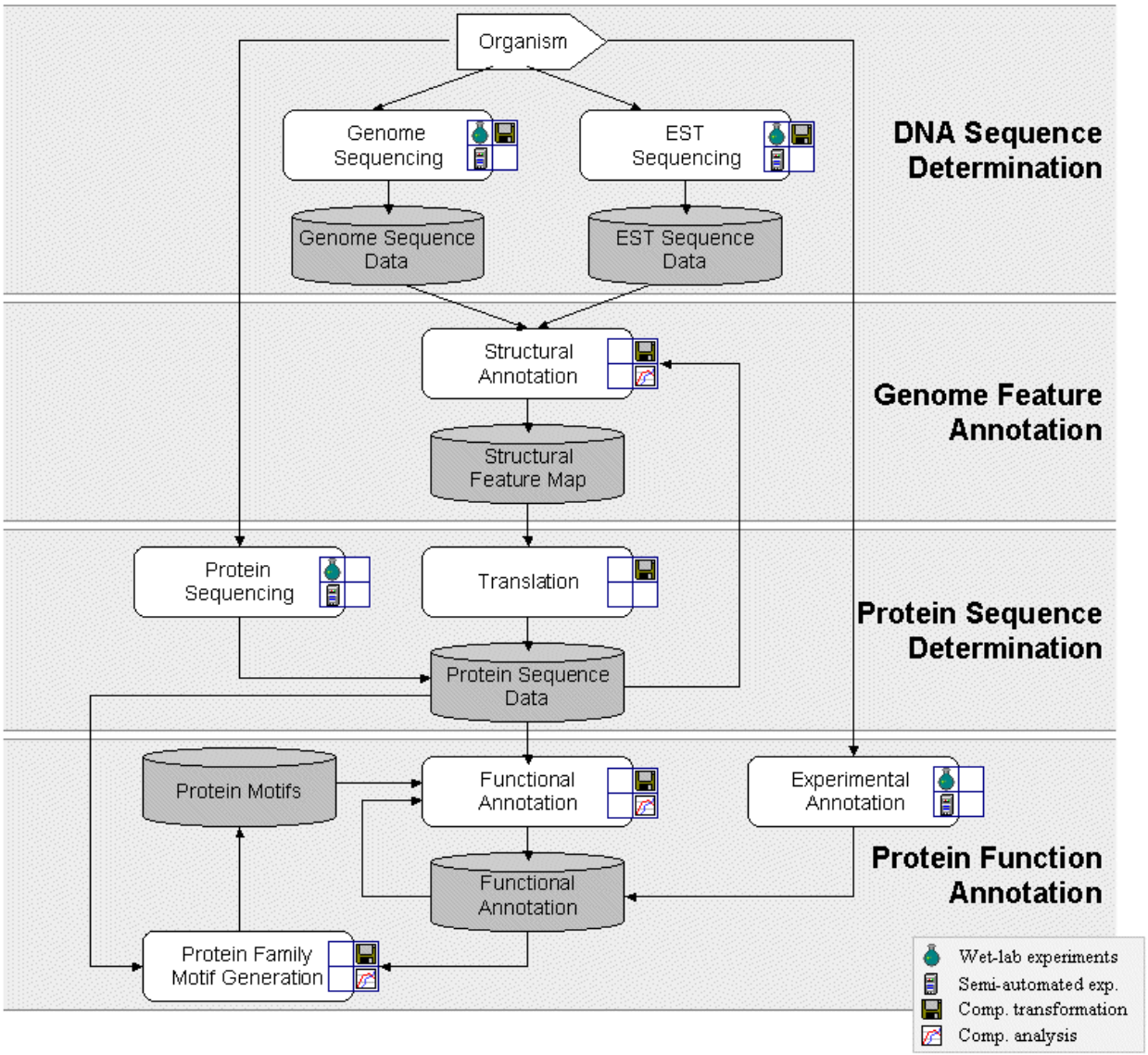


Figure 3-2: Genome data production is composed of four (mostly) dependent steps DNA Sequence Determination, Genome Feature Annotation, Protein Sequence Determination, and Protein Function Annotation. For each activity within these steps the data producing techniques are shown.

3.3 Errors in Genome Data Production

Apart from anecdotal evidence and our own experience working on some of the major life science databases, there are studies that show the existence and propagation of errors in genome databases. In [GABF00] the authors investigate the accuracy of several computer programs for predicting the structure of protein coding genes (structural annotation as explained later). None of the programs reaches an accuracy of 100% thus yielding inaccuracies in databases containing structural annotation. The maximum accuracy that might be achieved using currently available prediction programs investigated in [GABF00] is believed to be 90% for protein coding regions, and 70% for gene structure. Several studies show the existence of errors in functional annotation of proteins (explained later) [Bre99, DV01, ITA+03, Lin03]. In [Bre99] the error rate is estimated to be over 8% for annotations for the proteome of *Mycoplasma genitalium*. With the increased dependency on automatic annotation methods – due to the high data volume – this rate of errors is expected only to rise. In [DV01] the expected level of invalid data varies from less than 5% to more than 40%, depending on the type of annotated function. The authors estimate these numbers based on (a) the observation that most of the functional annotations are justified by relatively weak sequence similarities, and (b) the considerable number of discrepancies in functional annotations for sequences with high similarity. By extrapolating the discrepancies detected at a certain level of similarity to the number of proteins, the number of incorrect annotated protein function is estimated. *Linial* investigates functional annotations for short protein sequences (between 10 to 75 amino acids) in the draft of the human genome [Lin03]. Short sequences often lack statistical significance in similarity search programs, a method heavily used for functional annotation. Many of the short sequences are therefore termed 'hypothetical'. A brief survey among short hypothetical human protein sequences by the author suggests that for about two-thirds of them the correspondence to an actual protein of that length cannot be confirmed. However, at least in one case such an incorrect annotation already infiltrated the annotation of the rat genome. Finally, in [ITA+03] the authors generate a highly reliable set of annotations by carefully using automatic methods and experimental evidence. They compare their results with existing annotations and with the results of solely automatically performed annotations. For the original annotations only 63% of functional assignments within both datasets are in total agreement, while for the solely automatic annotations the precision is estimated to be 74% for the most reliable set of predictions.

In general, from the description of the genome data production process we can define several classes of errors within genome data:

- **Experimental errors** due to unnoticed experimental setup failure or systematical errors,
- **Analysis errors** due to misinterpretation of information,
- **Transformation errors** while performing transformations of information from one representation into another or one medium to another , e.g., data input,
- **Propagated errors**, when erroneous data is used for the generation of new data, and
- **Stale data**, i.e., unnoticed changes to base data on which a data item depends and that falsify it.

For the special case of errors in protein function annotation the TABS STANDARD (Transitive Annotation-Based Scale, [PG02]), defines classes of errors as (listed in descending order of gravity for error propagation): False positive, over-prediction, domain error, false negative, under-prediction, undefined

source, and typographical error. This classification is oriented towards the actual data, while our classification stems from the analysis of the data production process.

3.3.1 DNA Sequence Determination

DNA sequence determination starts from individual organisms and comprises the two parts DNA sequence determination and EST sequence determination. We ignore the second part for brevity. In DNA sequence determination, after isolating the DNA molecules from the cells, they are split into overlapping parts of about 1,000 bases, and then the sequence is determined for each of the parts using sequencing automata and software programs (*base calling*). Afterwards, the resulting sequence strings are input into an assembly program, which produces a representative sequence of the entire genome as a textual string.

Errors

The main types of erroneous information are incorrect sequence data and property values. They are both caused either by experimental errors or by transformation errors.

Experimental errors: The quality of sequence data mainly depends on the sequence preparation step and the experimental setup, as well as on the base composition of the DNA to be sequenced. Especially DNA regions containing high amounts of bases `c` and `g`, e.g., …`gcgagtgcgacgttcg`…, are difficult to sequence, because of physical constraints. In regions of repeating bases, e.g., …`gatggtgaaaaaaaaa`…, there is the possibility of missing a base because of overlapping signals. Poor experimental practice and improper use of chemicals can cause sample contamination or preparation failure.

Transformation errors: In the beginnings of DNA sequencing base calling has been an error-prone step. This has been improved with the use of modern high-throughput sequencing automata. In [Ric98] the error rate in sequences for six different sequencing projects is estimated between 0.23% and 2,58%. In sequence assembly, segments of DNA with near-identical sequence (segmental duplications), accounting for ~5% of the human genome, can result in sequence miss-assignment and wrong assembly of the sequenced parts. It is estimated that ~1.3% of the overall sequence of the June 2002 human genome draft sequence are erroneous due to assembly errors [CEK+03].

Quality Checks and Data Cleansing

Reliable quality checks in DNA sequence determination can be performed only after base detection. However, only fatal experimental errors are detected by searching for abnormal output display characteristics. AUTOEDITOR is a tool that improves base calling accuracy by utilizing information from an assembly of a genome [GSS04]. AUTOEDITOR considers the set of base calls aligned to a given position in the assembly and decides whether the base calling for this position has to be corrected or not. Application of AUTOEDITOR to recent genome sequencing projects shows that the number of erroneous base calls in these projects was reduced by 80%. A different sort of error in DNA sequences are *frame-shifts*, i.e., missing or inserted bases in the sequence string. When translating these sequences, the resulting protein has a completely different sequence starting from the position where the frame-shift occurs. Detecting frame-shift errors is probably the oldest data cleansing technique for genome data. Existing techniques are based on database lookup as they use protein databases to identify protein sequence that only partially align with the translated DNA sequence [Cla93, PR92]. Detection of frame-shift errors by sequence alignment is only possible when related sequences are found in the

databases. To avoid this limitation, *Fichant and Quentin* developed a method based on known intrinsic properties of coding sequences, like distribution of codons in the three frames of a DNA sequence [FQ95]. Based on such statistics, abnormal sequences are identified that point towards possible frame-shift errors. Erroneous DNA sequences may also result from contaminations. LUCY is a tool developed at THE INSTITUTE FOR GENOMIC RESEARCH (TIGR) that detects and removes contaminations [CH01]. The tool compares a newly determined DNA sequence with subsequences of common sequence polluters, like cloning vectors, to detect their occurrence in the determined sequence. In conclusion, while methods exist for detection of errors in DNA sequences, the most reliable sequences still result from sequencing each part multiple times.

3.3.2 Genome Feature Annotation

Genome feature (or structural) annotation results from performing a set of operations on the genome sequence data, e.g. sequence alignment or pattern search, making use of existing genome data and their annotations, e.g. aligning EST sequences against the genome to identify transcribed regions. Interpretation and combination of the results is guided by expert knowledge in form of *annotation rules*. These rules form the *annotation pipeline*, i.e., the description of the information production process. Often alternative ways for genome feature annotation are used, depending on the expert's preferences.

Errors

Errors in genome feature annotation include analysis errors, propagated errors, and stale data. They result in incorrect structural annotations.

Analysis errors: Incomplete or uncertain domain knowledge or careless interpretation of operation results can lead to misinterpretation and erroneous annotations. For example, predicting genes by simply using the occurrences of start/stop-codon-pairs results in a high number of wrongly predicted genes.

Propagated Errors: Errors in the genome sequence or genome data used within the annotation pipeline are propagated through the pipeline and result in misinterpretations and annotation errors later on. Sequence errors imply non-existent patterns or miss existing ones. Errors within additional data, e.g. EST sequences, can lead to operation results that cause erroneous interpretations.

Stale data: Annotation based on outdated data yields results different from annotations based on current data, causing inconsistency. The fact that changes to data items for the most part remain unnoticed by the depending data items is a major problem within genome data.

The errors in genome feature annotation are further classified as:

- **false positives**, e.g., parts classified as gene which are not coding for a protein,
- **false negatives**, e.g., parts not classified as gene which are coding for proteins, and
- **Incomplete or partially (in-) correct information**. This information, e.g., the uncertain start codon of a gene, is still included in several databases to avoid information loss. For example, in ENSEMBLE (Version 7.29, [HBB+02]) 36.77% of the predicted transcripts were incomplete [Mül03].

Quality Checks and Data Cleansing

Quality checks are performed only marginally within the process. There is the possibility of defining integrity constraints that have to be satisfied by the resulting data (see Section 3.4.2 for a case study in constraint repair in genome databases). However, not many helpful constraints are known, many have exceptions, and constraints are often not enforced to avoid information loss, or because constraint checking has to be performed by manual inspection or complex programs using additional data sources. Another quality checking method is to mine for errors, i.e., to detect outliers within the feature data, e.g., genes that are abnormally short or long.

3.3.3 Protein Sequence Determination

Protein sequence determination is performed experimentally, computationally, or by a combination of both. Computational protein sequence determination translates the predicted gene sequences using the genetic code of the specified organism. The protein sequence can also be determined experimentally making it independent of the two other steps performed before. Protein sequencing is hard to automate and therefore computational sequence translation is the preferred method. In some cases, a combination is used by determining a starting sequence (prefix) of the protein experimentally that is then used to search existing protein or translated DNA databases for proteins matching the prefix exactly.

Errors

The classes of errors resulting from protein sequence determination are experimental errors for experimental sequence determination and transformation errors, propagated errors, and stale data for computational sequence determination.

Experimental errors: As for DNA sequence detection, experimental errors result from poor experimental setup practices, or from failure of chemical reactions within the process.

Transformation errors: Using the wrong genetic code within the translation step, caused for instance by erroneous organism specification, results in an incorrect string representation of the actual protein sequence.

Propagated errors: Incorrect DNA sequences or frame-shifts result in incorrect translated protein sequences. Incorrect structural feature annotation yields false positives, i.e., translated proteins that are non-existent in the organism. Incomplete or partial information results in incomplete protein translations.

Stale data: Changes to the DNA sequence of the translated gene have to be reflected in changes to the resulting protein sequence. As those changes often remain unnoticed, the translated protein sequences become erroneous.

Quality Checks and Data Cleansing

Quality checks may be performed on the resulting protein sequence. Here, we can search automatically for proteins of uncharacteristic length or for unusual amino acid patterns within the protein sequences. This covers only very few errors. Reliable and efficient checking of correct sequences would require inexpensive, fast, and automated protein sequencing methods.

3.3.4 Protein Function Annotation

Protein function annotation is either performed computationally or experimentally. Computational annotation is based on the fact that the protein sequence determines the protein function. There are two main techniques for computational annotation of proteins. The first technique is based on protein similarity. It is assumed that two proteins with similar sequence very likely possess the same function. When finding annotated proteins in existing databases that are similar to a given query sequence, the annotated function is transferred from the sequence in the database onto the query sequence. The second technique searches for the occurrence of motifs in a protein. Each motif has an annotated function that is assigned to the query protein in case of motif occurrence. In many cases the two techniques are combined in annotation pipelines.

Experimental protein function annotation is much more reliable but also much more time consuming than computational annotation. A typical experimental annotation method is to generate genetically manipulated organisms not containing the gene for the protein under consideration and to observe how the behavior or the phenotype of the organism changes.

Errors

Experimental errors, analysis errors, propagated errors, and stale data are the classes of errors within this step, again depending on the method used.

Experimental errors: Due to the numerous experimental techniques for experimental protein function annotation, there are correspondingly oodles of possible errors that yield improper annotation of protein function.

Analysis errors: A major problem within protein annotation using sequence similarity is to define a sequence similarity threshold that allows sequences to be considered as having identical function. In [Doo87] it is stated that sequence similarity above 25% for proteins having minimum length of 100 amino acids is sufficient, while a similarity below 15% does not allow annotation transfer. In the interval between 15% and 25%, the proteins may very well be related, but additional studies must be performed to achieve higher confidence. Furthermore, similarity might not be present in the region that is responsible for the actual function of the annotated protein. Unfortunately, the responsible region is not always explicitly annotated. Thus, transferring function is error-prone. Insufficient inspection and careless usage of similarity search results very easily and very often lead to erroneous annotations.

Propagated errors: The huge amount of protein data produced by computational translation requires the application of computational annotation. Often, annotations are not marked as putative and used carelessly by other biologists. Incautious usage of putative annotations is responsible for a large degree of erroneous annotations.

Stale Data: As for propagated errors, stale data causes problems because of the high degree of data dependency for the results of the annotation process and because of changes to the annotation of proteins. This happens frequently as computational annotations are reproduced and changed due to experimental verification.

According to the classification of errors in genome feature annotation, errors in protein function annotation are divided into false positives and false negatives. False positives are annotations that were mistakenly assigned to a protein. False negatives are annotations that should have been assigned to a protein but were not [KL05].

Quality Checks and Data Cleansing

Quality checks involve performing additional studies to collect arguments for or against the correctness of an annotation. Unfortunately, these checks are often not performed, and thus most annotations have low confidence. Verification of protein function annotation requires the tedious documentation of the annotation process and results of the operations performed and decisions made. Such documentation is missing in most of the databases managing protein function annotation. Over the past couple of years, statistical methods have been designed that help identify erroneous protein function annotations. *Kaplan and Linial* present a protein-clustering method that enables automatic separation of false positives from true hits [KL05]. Their method quantifies the biological similarity between pairs of proteins by examining each protein's annotations, and then proceeds by clustering sets of proteins that received similar annotation into biological groups. Using a set of 327 test cases that are marked false positives the authors show that their method successfully separates false positives in 69% of the cases.

In [WKA04] the authors present XANTHIPPE, a post-processing system based on a simple exclusion mechanism and a decision tree approach. Their approach uses the C4.5 data-mining algorithm to filter erroneous annotations for protein sequences. Rules that are derivable from the learned decision tree are then applied to predicted, imported, and literature curated annotations of UNIPROT entries [UNI07]. For example, bacteria do not possess nuclear proteins because of their lack of a nucleus. Therefore, a 'Nuclear protein' keyword annotated on a bacterial protein is wrong disregarding the origins of the annotation, which could be predictive systems, data imports or even human curation. This constraint can be expressed as a simple exclusion rule, which if applied on the TREMBL section of UNIPROT not only removes 66 wrong keyword predictions produced by automated annotation, but also spots the same error in some imports from other databases (e.g. in the bacterial protein Q93HH7).

3.4 Genome Data Cleansing

Missing, incomplete, duplicate, or inaccurate information hampers automatic processing and analysis of data. Experiments based on poor quality data yield incorrect results. Such deficiencies lead to a loss in confidence in the underlying data source or the provider of the data, and to a rise in effort and frustration for the biologist on a day-to-day basis. Incorrect data may also lead to serious health consequences. Research in pharmacogenomics will enable pharmaceutical companies to produce drugs specifically designed against the genotype of individual patients. For such medications, incorrect data about the patient or about the drug compound and the genomic processes the compound affects may lead to serious consequences regarding the health of the so-treated patient. As we have shown, genome data is erroneous by nature – due to its production process. The data is produced by experiments that are error prone and analyzed by domain experts in a subjective manner using uncertain knowledge leading to invalid, uncertain, or incomplete data. Data dependencies inherent to the production process and to the usage of the data make genome data predestined for propagated errors. Furthermore, there are frequent changes in the data and knowledge that in many cases remain unnoticed to systems storing derived data. While there has been much research in developing a general data cleansing framework, and while many data cleansing methods and applications have been developed for certain domains, such as health-care data [LS02, PG02], there is yet little research addressing the particular, and novel data cleansing problems as they occur in the life sciences domain.

3.4.1 Challenges in Genome Data Cleansing

Two approaches could eliminate many of the data quality issues raised in this chapter at production time. First, to keep pace with the analysis of the huge amount of data produced, reliable methods for genome data production could be employed, i.e., using repeated experimental methods and less automation for data analysis. Second, quality checks within the production process could be employed. Quality checks are often omitted, because they usually require manual inspection and the huge amount of data makes them time-consuming and expensive. Within the domain of genome data there are only few reliable constraints and a multitude of exceptions hinder effortless verification of data correctness. The multitude of different sources necessary for result comparison and verification poses another problem with genome data. There is no standard format for genome data storage and no commonly accepted vocabulary. This hampers integrated access and makes data transformations for standardization and normalization necessary.

The afore-mentioned reasons make data cleansing a necessity for genome data after data production. Existing cleansing approaches are mainly concerned with producing a unified and consistent data set, addressing primarily syntactical problems and ignoring the semantic problem of verifying the correctness of the represented information. Duplicates also exist in genome data, however, duplicates are less interfering than in other application domains. Duplicates are often accepted and used for validation of data correctness. In conclusion, existing data cleansing techniques do not and cannot consider the intricacies and semantics of genome data, or they address the wrong problem, namely duplicate elimination. We see three concrete and reasonable challenges for genome data cleansing.

Credibility checking and re-annotation

The most reliable way for semantic genome data cleansing is to re-perform experiments and computational analysis under careful control by domain experts. This procedure is time-consuming and expensive and it is also performed for already correct values, yielding a large amount of unnecessary computation and experiments. Credibility checking on data to identify those yielding evidences for being erroneous and re-annotated them can reduce the overall cost of re-annotation. Credibility checking is also important to verify the correctness of data before it is used within other processes. Arguments for or against correctness of data are generated by domain dependent evidence functions or integrity constraints. Evidence functions operate on existing data and check known biological facts and rules. For example, certain amino acid combinations are known to be non-existent in proteins or the evidence for a predicted coding region is high if similar regions exists within other organisms, i.e., the sequence region is conserved. The methods used within the evidence functions and for constraint checking are the same as those within the annotation pipelines.

Metadata management

One of the main problems in verifying data correctness is the missing metadata about how the data was gained and what other information and interpretations it is based upon. The information used within the production process of derived data is called the *data lineage*. Data lineage can be used for keeping annotations up-to-date in a changing environment without completely re-annotating the whole database every time parts of the data used for annotation changes. A data cleansing framework for genome data has to be able to detect and react on changes in the base data without re-performing the complete and expensive data cleansing process. By using the data lineage the data items that depend on the changing data are easily identified for re-annotation.

Alternative solution management

In data cleansing it is often impossible to find a correct solution immediately. Instead, there often exists a set of alternative solutions. These solutions have to be managed up to a point in time when one is able to decide which one is the correct value. Until then, the alternatives have to be included within the process of further data production. After deciding which is the correct solution from a set of possible solutions, one has to be able to undo decisions that were based on data that has become obsolete now. For this purpose data lineage information is used to identify depending data items.

3.4.2 Genome Data Cleansing using Integrity Constraints

Our discussion of quality checks and cleansing abilities in Section 3.3 shows that mainly statistical methods from the set of cleansing methods (presented in Section 2.4) are used for genome data cleansing. In [Mül03] we performed an experimental study on semantic data cleansing for sequence annotation data from the ENSMBL database [HBB+02], using an integrity constraint to identify data deficiency and re-annotation to derive new values for incorrect ones. The main steps of the cleansing process are outlined in Figure 3-3.

The integrity constraint we used in our study is "*The translation of RNA always starts at the codon 'ATG'*". Using the MYSQL load files for ENSEMBL database (Release 7.29) we installed a local copy of the relational database in our IBM DB2 database system and checked the above mentioned biological constraint. Error detection is done using a simple SQL query filtering those translations starting with a codon different from 'ATG' (Step 1). Using protein sequences imported from the ORACLE dump-file release of SWISS-PROT/TREMBL [BBA+03] (released July 15, 2002)[1], we defined a re-annotation function which calculates the correct start codon using automatic processing (Steps 2-3). For re-annotation of wrongly annotated translation starts, we first translated the upper end of the corresponding transcript into the according protein sequence. We then aligned protein sequences from SWISS-PROT/TREMBL against the translated transcript (Step 2). If such an alignment exists, the left end of the aligned sequence marks the position of the new start codon when aligned against the original DNA (Step 3). Aligning DNA and protein sequences requires translation of the DNA sequence from the left of the annotated start codon into the according amino acid sequence. We thereby allow a limited number of replacements, insertions, and deletions in the alignment. Note that the codon 'ATG' is always translated into the amino acid *Methionine*. Therefore, we only used those sequences from SWISS-PROT/TREMBL for the alignment in the second step that start with amino acid *Methionine*.

About 30% of the translation entries in the ENSEMBL release violated the integrity constraint used in our study. For nearly 15% of these violating entries a new start codon was proposed by our re-annotation function. A short survey of the ensuing releases of ENSEMBL and SWISS-PROT/TREMBL showed that the database curators updated some of the identified corrections, enabling us to validate our methods. Our study shows the applicability of domain specific integrity constraint repair for genome data cleansing. However, our solution cannot be generalized and required significant programming effort instead of using 'of the shelf' cleansing methods. The same restriction on usability of existing tools and methods appears to be true for many other genome cleansing approaches mentioned in Section 3.3, making re-annotation a programming intensive task.

[1] see ftp://ftp.ebi.ac.uk/pub/contrib/swissprot/oracle/README.html for more details

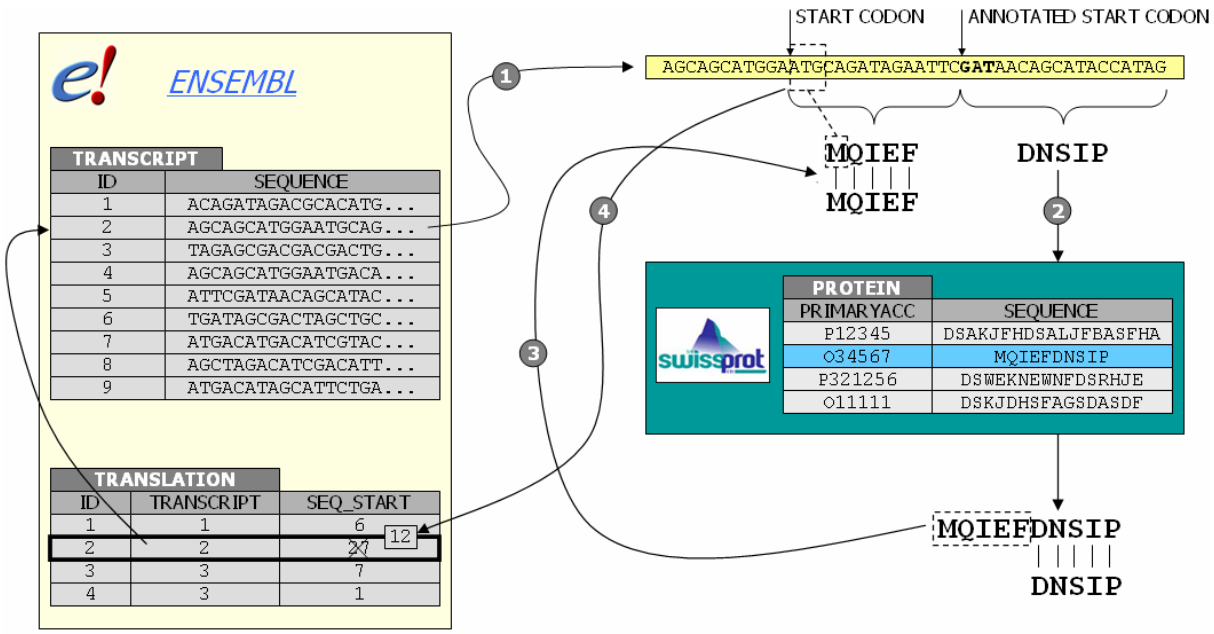


Figure 3-3: The four main steps of cleaning translation starts in ENSEMBL involve (1) extraction and translation of the DNA sequence, (2) alignment of the translated protein with entries in SWISS-PROT/TREMBL, (3) alignment of the prefix of matching entries with the extracted DNA sequence, and (4) correction of translation starts in case of identified proper start codons.

3.4.3 Genome Data Cleansing using Overlapping Data Sources

A second approach to genome data cleansing utilizes the huge amount of additional and redundant data in different sources. By detecting and highlighting contradictions or accordance, one collects reasons for or against the correctness of data items. Existing overlapping data sources may then be merged to form a single consistent view on the data. We use COLUMBA[2] as our motivating example for merging overlapping databases for data cleansing purposes. COLUMBA is a database of integrated protein annotations that is developed and maintained in a joint effort by different research institutions affiliated in the BERLIN CENTER FOR GENOME BASED BIOINFORMATICS (BCB) [RMT+04, RTM+05]. COLUMBA physically combines resources on protein structures into a single relational database. The main source for protein annotations in COLUMBA is the PROTEIN DATA BASE (PDB) [BKW+77]. In COLUMBA, we pay special attention to the aspect of measuring data quality and detecting hot-spots of poor quality in these annotations. We approach the problem by analyzing contradicting values in the case of duplicate protein entries. In COLUMBA, such duplicates do not appear at the data sources, which are considered independent, but in the core data, i.e. the PDB entries, itself. Currently the PDB database is available in three different versions and formats: (i) the original PDB data available in flat file format, (ii) data in MACROMOLECULAR CRYSTALLOGRAPHIC INFORMATION FILE FORMAT (MMCIF) from the PDB uniformity project at the UNIVERSITY OF CALIFORNIA SAN DIEGO (UCSD) aiming at removing inconsistencies in PDB data [BBF+01], and (iii) the MACROMOLECULAR STRUCTURE RELATIONAL DATABASE (E-MSD), a comprehensive cleansing project at the EUROPEAN BIOINFORMATICS INSTITUTE (EBI) to ensure data uniformity and to create a single access point for protein and nucleic acid structures and related information, available as ORACLE dump files

[2] http://www.columba-db.de/

[BDF+03]. The latter two independent cleansing projects focus on different quality aspects of the original PDB data. By merging data from these sources, we aim at generating a resulting high quality set of protein structure annotations. We currently utilize the parser for PDB flat-files to create an instance of our PDB target schema and the OPENMMS TOOLKIT, containing software for parsing and loading MMCIF files into a relational database. This toolkit uses a complex schema consisting of approximately 140 tables. We generated a set of schema mapping rules, which transform the data from the OPENMMS schema into a simpler target schema comprising only 6 tables. Thereby, we are able to create two overlapping instances for our PDB target schema, referred to as PDB and OPENMMS in the following.

For COLUMBA we face the non-trivial problem of having to choose the best origin for each single attribute value from the overlapping instances. Identification of corresponding records within the two instances is easily done via the unique PDB identifier forming a matching record pair. We analyze these matching pairs for mismatches in their attribute values. The percentage of mismatches within the attributes varies widely (see Figure 3-4). Attributes having close to 100% mismatches often result from different formats or NULL values within one of the instances, leading to straightforward conflict resolution strategies. Further investigating the mismatch causing values within each of the attributes reveals additional information about the causes for the mismatch. For instance, comparison and evaluation enabled us to identify 32 records having a deposition year of '1900' in the MMCIF files where the original PDB flat files state the year '2000' for entry deposition. In another case, the structure method for over 2000 records resulting from parsing the PDB flat files was 'unknown' while the mmCIF files stated 'X-ray diffraction' as the structure method used. In general, by comparing the overlapping instances and highlighting and evaluating the differences using domain knowledge, we are able to identify the reliable parts within both instances to select them for integration into a resulting instance of the PDB target schema. In the remainder of this thesis, we describe in detail the algorithms used for revealing regularities in contradicting data.

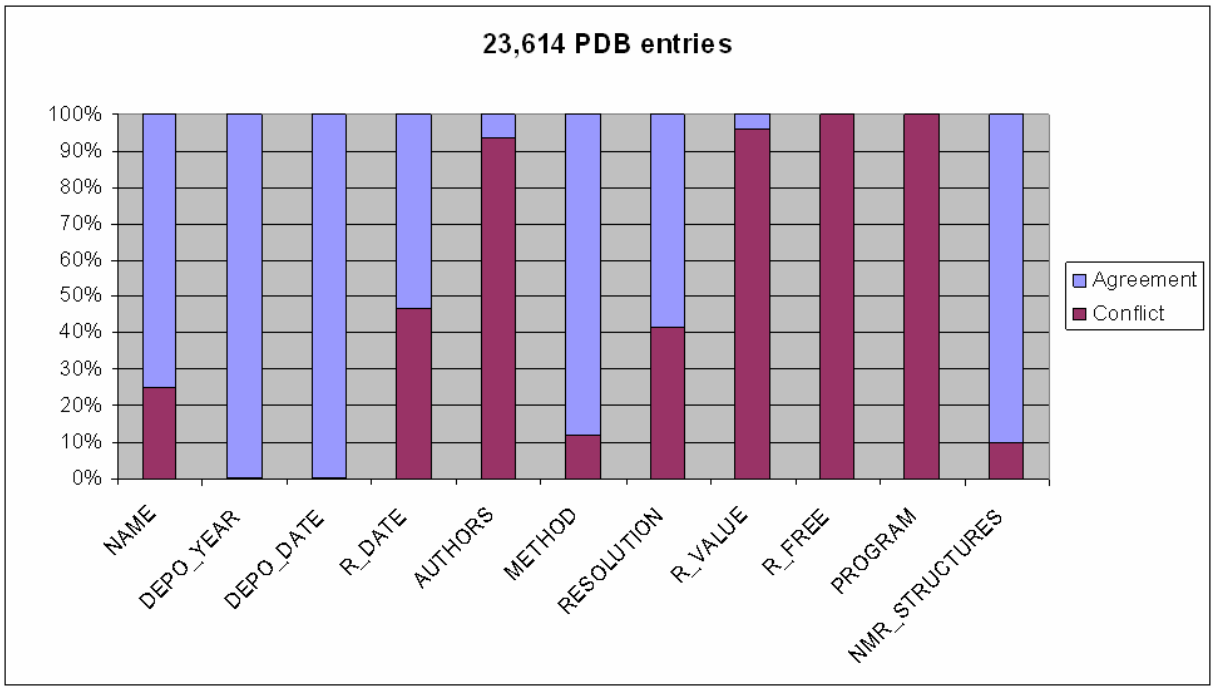


Figure 3-4: Statistic of conflicts for different attributes in PDB entries between data in the two instances PDB and OPENMMS from the COLUMBA project.

3.5 Summary and Related Work

Genome data is dirty and this state is caused by inadequacies of the data production process. We present typical cases of errors extracted from the literature together with quantifications for these errors [Bre99, CEK+03, DV01, ITA+03, Lin03, Ric98]. Some of the given examples emanate from our own experiences. We give a classification of quality problems in genome databases derived from a description of the genome data production process. We also give reasons why errors cannot be avoided simply by changing parts of the production process. Only prohibitively expensive quality checking within the process, using quality checking modules, can increase quality during data production. This leads to the problem of how to eliminate existing errors through data cleansing methods. Most of the existing cleansing methods described in Section 2.4 are not applicable for the major errors found in genome data. The AJAX cleansing framework has recently be adopted for biological data [HGP+04]. However, the resulting BIO-AJAX framework simply adds a classification operator to the set of operators defined in [GFSS00, GFS+01]. The classification operator allows categorizing data according to domain rules. BIO-AJAX has been applied to the nomenclature problem in TREEBASE, a phylogenetic and evolutionary information system, to solve inconsistency problems among taxonomy names. In this chapter, we list related work for detecting and eliminating incorrect values in genome data for each step of the defined genome data generation process. Our discussion shows that methods for eliminating deficiencies currently exist primarily for DNA sequence data [CH01, Cla93, FQ95, GSS04, PR92, STR+03]. For annotation data, existing approaches to identify erroneous annotations are mainly focused on protein function annotations [KL05, WKA04]. Elimination of incorrect annotations is done by excluding the identified incorrect values.

In general, re-annotation of incorrect values is a possible approach to increase genome data quality. Just like annotation itself, re-annotation requires domain dependent evidence functions. The definition of a set of general evidence functions for the domain of genome annotation will enable us to build a formal model to specify the annotation and cleansing process. The intrinsic properties of these individual functions can be used to detect erroneous annotations without the necessity of complete re-annotation. There are tools being developed that allow the specification of annotation pipelines or workflows without massive programming overhead, e.g., BIOPIPE [HRC+03] or KEPLER [LAB+05]. However, these tools still rely on an expert user being able to define a re-annotation process that produces more reliable annotations than the existing ones. In those cases where alternative solutions and evidence values are managed, it is desirable to include them within the annotation and cleansing process to receive results of higher quality. Some of the genome databases are currently beginning to manage such evidences for their entries. Credible annotations can be derived by excluding invalid or unreliable entries from the processing. The formal model for genome annotation has to take these evidences into account. Including the management of annotation lineage within the model further enables efficient detection and re-annotation of affected annotations when changes in external data sources occur. Integration of overlapping data is another option for genome data cleansing. When merging databases there is also the need for assistance by an expert user. However, in this case cleansing relies on existing data that results from already performed annotation processes.

Part II

Mining Contradictory Data

Chapter 4

Mining for Patterns in Contradictory Data

Overlapping databases are valuable sources of information for semantic data cleansing, provided that we are able to identify and resolve conflicts effectively. Within this chapter, we describe an algorithm for comparing pairs of overlapping databases. The algorithm specifically searches and finds interesting conflicts in this comparison, i.e., conflicts that occur in some sense systematically or follow certain patterns. We call those cases *contradiction patterns*. Contradiction patterns describe regularities in conflicts occurring together with certain attribute values, or with other conflicts. Contradiction patterns are a very quick way to find quality hotspots in two data sets, since they help to ignore spurious problems. On the other hand, these patterns give a human expert who has the necessary domain knowledge valuable clues to reasons for inconsistencies. In this sense, contradiction patterns identify groups of conflicts that potentially follow the same systematic conflict reason. These groups of conflicts form the basis for context-aware conflict resolution.

The contradiction patterns we find are a special kind of association rules. Association rules are a popular concept for knowledge representation. Their simple structure eases interpretation and makes adopting association rule mining algorithms for mining contradiction patterns a natural choice. We start by introducing the general problem of association rule mining and give a brief overview of association rule mining algorithms. We then describe an algorithm for closed pattern mining that forms the basis for our work on contradiction pattern mining. The methods we present in this chapter were developed in the course of the COLUMBA project described in the previous chapter. In COLUMBA, we quickly found that the amount of inconsistencies is overwhelming and that we therefore need to focus on the most interesting (= most annoying) inconsistencies. Using the algorithm presented in this work, we achieved this focusing, which lead to the detection of various parser errors, different understanding of data in the two cleansing projects, and, especially, truly conflicting data. In Section 4.2, we present a simple model for conflicts between overlapping databases. In Section 4.3, we describe an algorithm for finding contradiction patterns. The presented algorithm is an extension of our work on contradiction pattern mining presented in [MLF04]. We discuss enhancements to the original algorithm. In our experiments in Section 4.4, we show how variations to the interestingness measures used in contradiction pattern mining affects the number of identified patterns for overlapping databases for protein structure annotations.

4.1 Association Rule Mining

Association rule mining has become the most popular data mining technique in databases, being reflected by awarding the VLDB TEN YEAR BEST PAPER AWARD to *Rakesh Agrawal* and *Ramakrishnan Srikant* in 2004 for their work on '*Fast Algorithms for Mining Association Rules in Large Databases*' [AS94, NÖK+04]. Initially, association rule mining was defined for customer transaction data to assist business management by providing information about sets of items that tend to be purchased together by customers [AIS93]. A typical example is a statement that "*90% of customers that purchase bread and butter also purchase milk*". This information is useful for making decisions about shop layout or planning of special sales offers. Association rules for customer data are also used in Customer Relationship Management (CRM) for cross-selling, i.e., recommendation of products to customers based on the items they are interested in and association rules in the form of "*People who were interested in these items also liked ...*" [BSVW04]. Over the last years, association rule mining techniques have been adopted for various other data types and purposes, like spatial data [KH95, SV00], temporal or event sequence data [AS95, MTV97, NYC06], manufacturing process control [Kus02], biological and medical data [ACT05, OES06, OSB00], legal decision making [GS01], or the afore mentioned data cleansing purposes [HH04, KLK+04, MM00]. We restrict our description of association rule mining algorithms to these traditionally used for customer transactions in supermarkets.

4.1.1 Formal Problem Definition

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of distinct literals, called items, that represent, for example, the items being available for purchasing in a supermarket, e.g., milk, butter, and bread. A set of items $X \subseteq I$ of size $k = |X|$ is called a *k-itemset* or simply an *itemset*. A transactional database D is a multi-set of subsets of I, called transactions T, i.e., $D = \{T_1, T_2, ..., T_m\}$ with $T_j \subseteq I$, $1 \leq j \leq m$. Conceptually, a transaction T models a customer's purchase or shopping cart with each item $i \in T$ being an item purchased by the customer. Each transaction has a unique identifier, called *transaction id* (*tid* for short). We refer to individual transactions by T_{tid}. For a single item i the tid-list, denoted by *tidlist*(i), is the set of transaction identifiers that correspond to the transactions containing this item. Accordingly, tid-lists also exist for every itemset X, denoted by *tidlist*(X), as the intersection of the tid-list of the items in X, i.e., $tidlist(X) = \bigcap_{i \in X} tidlist(i)$. An *association rule* is a probabilistic statement about co-occurrences of items in a transactional database. In general, an association rule takes the form:

IF X **THEN** Y **with probability** p, $X, Y \subset I$ and $X \cap Y = \emptyset$.

The meaning of an association rule is that if items X are bought in a transaction, items Y are being bought in the same transaction with probability p, i.e., p is the conditional probability $p(Y \subset T \mid X \subseteq T)$. An association rule is usually expressed in the form $X \Rightarrow Y$ with the left hand side being called the *antecedent* of the rule and the right hand side the *consequent*. The notation for association rules is sometimes misleading since an association rule is not a strict implication. Association rules are rather "probabilistic implications" where X implies Y with some probability p. Strictly speaking the term association rule is a misnomer since these rules are inherently correlational but need not be causal [HMS01].

From the definition of association rules it becomes clear that generally any pair of non-empty itemsets (X, Y) is considered an association rule with a certain probability. Therefore, the main challenge for association rule mining is the immense number of rules that theoretically must be considered. Since association rules are intended to support decision makers, there is a need to avoid flooding the user with rules that are irrelevant to them. Thus, numerous quality or interestingness measures for association rules have been developed to restrict the size of the returned rule set (see [PKS02] for a comprehensive overview and discussion of their strength and weaknesses). The most commonly used quality measures are support and confidence, as described by *Agrawal, Imieliński, and Swami* [AIS93]. The *support* of an association rule $X \Rightarrow Y$ is defined as the percentage of transactions in D that contain both, X and Y. In general, *sup(X)* denotes the percentage of transactions that contain an itemset X. The *confidence* of an association rule is defined as the ratio of transactions that contain X and Y over the number of transactions that contain X:

$$sup(X) = \frac{|\{T \mid T \in D \wedge X \subseteq T\}|}{|D|} \qquad conf(X \Rightarrow Y) = \frac{|\{T \mid T \in D \wedge (X \cup Y) \subseteq T\}|}{|\{T \mid T \in D \wedge X \subseteq T\}|}$$

Support conforms to the statistical relevance of an association rule while confidence conforms to the conditional probability $p(Y \subset T \mid X \subseteq T)$. Confidence is also called the strength or accuracy of an association rule since it defines the (statistical) accuracy of assuming Y when X is contained in a transaction T.

Example 4-1: Consider the transactional database D containing 10 transactions over a set of 8 items. The table on the right lists all itemsets Z_i that have support equal or above a threshold of 0.5, together with the list of transactions $tid(Z_i)$ that contain these itemsets. Itemset $Z_9 = \{i_2, i_3, i_4\}$ is contained in five out of ten transactions. The support of an association rule $\{i_2, i_3\} \Rightarrow \{i_4\}$ therefore is 0.5. The confidence of this association rule is 0.625, since the antecedent of the rule is also contained in transactions T_4, T_6, and T_{10}.

D

TRANSACTION	PURCHASED ITEMS
1	i_1, i_2, i_3, i_4
2	i_2, i_3, i_4, i_5, i_6
3	i_4, i_5, i_7, i_8
4	i_2, i_3
5	i_1, i_3, i_4, i_8
6	i_2, i_3, i_8
7	i_2, i_3, i_4
8	i_2, i_3, i_4, i_7, i_8
9	i_1, i_2, i_3, i_4, i_8
10	i_2, i_3, i_8

Frequent Itemsets having Support ≥ 0,5

Frequent Itemsets	Tid-list	Support
$Z_1 = \{i_2\}$	{1, 2, 4, 6, 7, 8, 9, 10}	0.8
$Z_2 = \{i_3\}$	{1, 2, 4, 5, 6, 7, 8, 9, 10}	0.9
$Z_3 = \{i_4\}$	{1, 2, 3, 5, 7, 8, 9}	0.7
$Z_4 = \{i_8\}$	{2, 3, 5, 6, 8, 9, 10}	0.6
$Z_5 = \{i_2, i_3\}$	{1, 2, 4, 6, 7, 8, 9, 10}	0.8
$Z_6 = \{i_2, i_4\}$	{1, 2, 7, 8, 9}	0.5
$Z_7 = \{i_3, i_4\}$	{1, 2, 5, 7, 8, 9}	0.6
$Z_8 = \{i_3, i_8\}$	{5, 6, 8, 9, 10}	0.5
$Z_9 = \{i_2, i_3, i_4\}$	{1, 2, 7, 8, 9}	0.5

■

The simple structure of association rules and the intuitive definition of support and confidence make association rules inherently understandable – even for non-experts in data mining. Coupled with their direct applicability to business problems, association rules have become a popular concept for data mining and knowledge discovery. The general problem of mining association rules is defined as follows: given a transactional database *D,* find the set of association rules having support and confidence equal or above given thresholds min_{sup} and min_{conf}. According to this problem statement the basic algorithm for mining association rules was introduced by *Agrawal, Imieliński, and Swami* in 1993 [AIS93] and modified by *Agrawal and Srikant* in 1994 [AS94] to scale for large transactional databases.

4.1.2 Frequent Itemset Mining

The support and confidence constraints, also referred to as the *support-confidence framework*, hold two important properties that benefit efficient association rule mining in large databases. First, there is a direct relationship between the confidence and the support of an association rule. The confidence of an association rule $X \Rightarrow Y$ equals the quotient of the sup($X \cup Y$) and sup(X):

$$conf(X \Rightarrow Y) = \frac{|\{T \mid T \in D \wedge (X \cup Y) \subseteq T\}|}{|D|} \cdot \frac{|D|}{|\{T \mid T \in D \wedge X \subseteq T\}|} = \frac{sup(X \cup Y)}{sup(X)}$$

This direct relationship between support and confidence allows algorithms for finding association rules to be decomposed into two sub-problems:

1. First, generate all subsets of items $Z \subseteq I$ that have support in the transactional database above the given threshold, i.e., $sup(Z) \geq min_{sup}$. Combinations of items that fulfill the support threshold are called large or *frequent itemsets*. Since every association rule $X \Rightarrow Y$ has to have support above the given threshold, there has to be a frequent itemset $Z = X \cup Y$.
2. Generate the desired association rules using the frequent itemsets: For each frequent itemset Z find all non-empty subsets W. Output an association rule $W \Rightarrow (Z - W)$ if the ratio of $sup(Z)$ to $sup(W)$ is at least min_{conf}.

Algorithms that find frequent itemsets Z usually determine the support for all subsets of Z in advance. Thus, having solved the problem of finding frequent itemsets, the solution to the problem of deriving association rules from these itemsets is straightforward. Existing work on mining association rules using the support-confidence framework therefore primarily focuses on the problem of frequent itemset generation, i.e., given a transactional database D find all sets of items that have a support equal or above a given threshold min_{sup}. This problem is referred to as *frequent itemset mining* or *large itemset mining*.

The computational complexity of frequent itemset mining steams from the immense search space that grows exponentially with the size if I. For a given set of items I the number of potential frequent itemset is $2^I - 1$. For large I - a number of 100 items is certainly not uncommon for today's large reseller markets - enumeration of all subsets is computationally infeasible, let alone counting the support for each one of them. To avoid enumeration of all possible subsets, mining algorithms utilize a second property of the support-confidence framework, that is, the support for an itemset Z cannot be greater than the support for any of its subset. This property, referred to as the *downward closure property of support* [AS94], ensures that if we find an itemset having support below a given threshold, there is no need to consider any of its supersets since their support cannot increase. The search space for frequent

itemsets forms a lattice as shown in Figure 4-1 [HGN00]. Due to the downward closure of support, the support threshold forms a border in the lattice that separates the frequent from the infrequent itemsets. The frequent itemsets are located in the upper part of the figure whereas the infrequent ones are located in the lower part.

Since the introduction of association rule mining in 1993 several different algorithms for efficient frequent itemset mining have be developed, e.g., [AS94, BMUT97, HKK00, HPY00, PCY95, Toi96]. The basic principle of common frequent itemset mining algorithms is to employ the border formed by the support threshold in the search space to efficiently prune the search space. The border is found, whenever an infrequent itemset is found while traversing the search space. *Hipp, Güntzler, and Nakhaeizadeh* give a systematization of frequent itemset mining algorithms together with some of the representatives for each class of algorithms [HGN00]. Existing approaches are classified a) regarding their strategy to traverse the lattice, and b) by their strategy to determine the support for itemsets. Traversal of the search space may either be done in breadth-first or in depth-first order. In a breadth-first approach, the support for all (*k-1*)-itemsets is determined before counting support values for the *k*-itemsets. In contrast, depth-first approaches recursively descend the tree until an infrequent itemset is encountered. There are two common approaches to determine the support of an itemset. One approach is to directly count the occurrences of an itemset in the database. While the database is scanned a counter for each itemset is increased whenever this itemset is recognized as a subset of the currently examined transaction. In [AS94] the authors propose a hash-tree data structure that enables efficient detection of all itemsets that are contained in a given transaction. A second approach for support counting of an itemset X is intersection of the tid-list of all the items in X. Sorting the tid-list in ascending order allows efficient intersection.

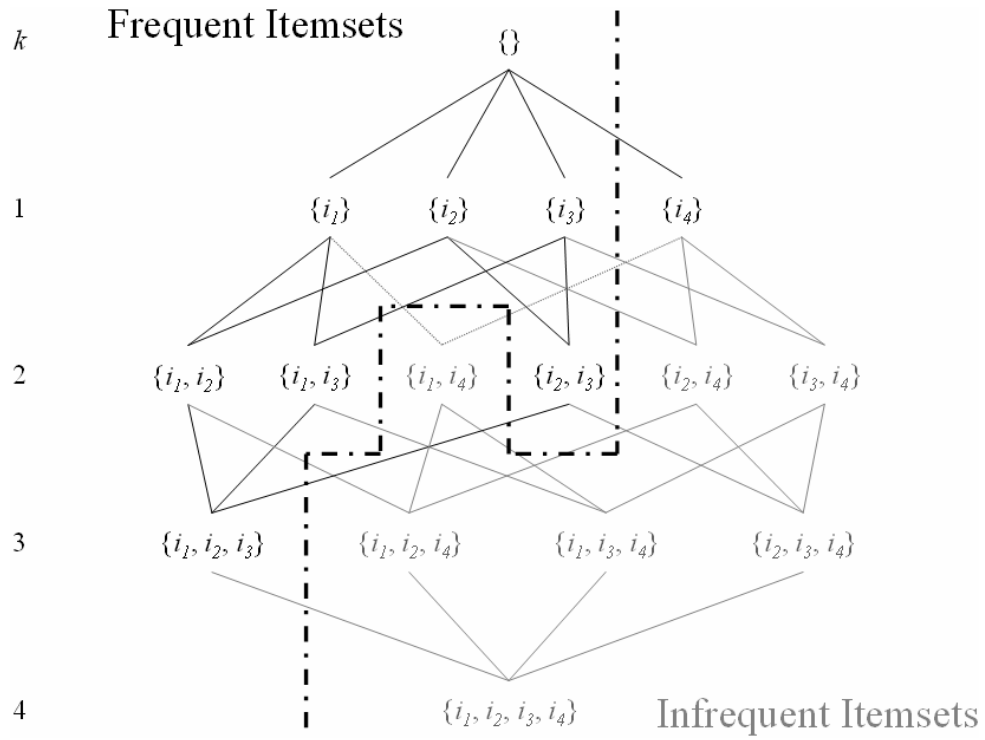


Figure 4-1: Visualized search space for frequent itemsets over a set of four items. The support threshold forms a border in the lattice. Frequent itemsets are located above the border and infrequent itemsets are located below it.

Apriori Algorithm for Frequent Itemset Mining

The APRIORI-algorithm as described in [AS94] traverses the search space in breadth-first order making multiple passes over the data. In the first pass, the support for each item is determined to form the set of frequent items. Furthermore, the set of frequent items forms the starting set of frequent *1*-itemsets. Each following pass starts with a seed set of frequent itemsets that were found in the previous pass. For the *k*-th pass the seed set of frequent itemsets consists solely of frequent (*k-1*)-itemsets. Itemsets in the seed set are extended before the next pass over the data to form potentially frequent itemsets, called *candidate itemsets*. Items in each itemset are ordered in lexicographical order. Extension is done by adding single frequent items to frequent (*k-1*)-itemsets that occur behind any of the items in the itemset in lexicographical order. These extensions form a superset of frequent *k*-itemsets. Instead of counting the support for each of the extended itemsets APRIORI deletes all *k*-itemsets for which at least one of their *k* (*k*-1)-subsets is not contained in the seed set of frequent (*k-1*)-itemsets. Thus, the algorithm concludes *a priori* to passing over the data that these itemsets cannot be frequent due to the downward closure property of support. The support for the remaining candidate itemsets is counted during a pass over the data. At the end of the pass it is determined, which of the candidates are actually frequent and these frequent itemsets become the seed for the next pass. This process is continued until no further frequent itemsets are found.

Theoretically, there is still an exponentially number of frequent itemsets. In practice, however, the number is much smaller, depending on the support constraint and the distribution of the data. In Example 4-1, we count the support for each of the 8 items in the first pass over the data. Considering a support threshold of 50% there are four frequent itemsets of length one, i.e., itemsets $\{i_2\}$, $\{i_3\}$, $\{i_4\}$, and $\{i_8\}$. In the second pass we then have to count support for the six possible candidates of length two, i.e., $\{i_2, i_3\}$, $\{i_2, i_4\}$, $\{i_2, i_8\}$, $\{i_3, i_4\}$, $\{i_3, i_8\}$, and $\{i_4, i_8\}$. Only four of these *2*-itemsets are frequent, i.e., $\{i_2, i_3\}$, $\{i_2, i_4\}$, $\{i_3, i_4\}$, and $\{i_3, i_8\}$. Using the resulting frequent itemsets the three candidate itemset $\{i_2, i_3, i_4\}$, $\{i_2, i_4, i_8\}$, and $\{i_3, i_4, i_8\}$ of length three are generated. Itemset $\{i_2, i_4, i_8\}$ cannot be frequent since subset $\{i_2, i_8\}$ is not a frequent itemset. After counting support for the remaining two candidates only itemset $\{i_2, i_3, i_4\}$ remains. The only possible extension, i.e., $\{i_2, i_3, i_4, i_8\}$, can again not be frequent due to $\{i_2, i_8\}$ not being frequent. Overall the algorithm counts support for only 16, i.e., 8 + 6 + 2, of the 2^{10}-1 possible subsets.

FP-tree - Frequent Itemset Mining without Candidate Generation

In [HPY00] *Jiawei Han, Jian Pei, and Yiwen Yin* presented a fundamentally new approach to frequent itemset mining without candidate generation. Candidate generation is costly, especially if there is a multitude of frequent items or long frequent itemsets. For example, if there are n frequent items, Apriori generates *(n - 1) + (n - 2) +...+ 1* candidate *2*-itemsets and counts support for them. Moreover, to discover a frequent itemset of size *n*, Apriori must generate each of the 2^n*-1* frequent subsets as candidates. To avoid the problem of candidate generation, *Han et al.* construct a compact data structure representing transaction data, called *frequent pattern tree* (FP-tree), which is an extended prefix-tree. Only frequent items will have nodes in the tree. If two transactions share a common prefix, according to some sorted order of frequent items, they share a prefix path in the FP-tree. The support count for each item is registered in the nodes of the FP-tree. If the frequent items are sorted in their frequency descending order, there are better chances that more prefixes can be shared. Thus, the tree nodes are arranged in such a way that more frequently occurring items will have better chances of sharing nodes than less frequently occurring ones.

Based on the FP-tree a pattern fragment growth mining method is developed (FP-GROWTH). The method starts from a frequent *1*-itemset as an initial suffix pattern and examines only its conditional pattern base, a sub-database which consists of the set of frequent items co-occurring with the suffix pattern, called the *conditional FP-tree*. Mining is performed recursively with such a tree. The itemset growth is achieved via concatenation of the suffix pattern with the new generated ones from the conditional FP-tree. Experimental studies show that FP-GROWTH is at least an order of magnitude faster than APRIORI. The margin becomes even wider when the frequent itemsets grow longer.

4.1.3 Closed Itemset Mining

In general, performance of frequent itemset mining algorithms deteriorates quickly for low support threshold or dense datasets. A simple example given in [WHP03] states that for a database having only one transaction of length 100, there exist 2^{100}-*1* frequent itemsets if the minimum support threshold is 1. The example also shows that itemsets are often redundant in that they describe the same set of database rows. These redundancies are not only obstructive regarding efficient itemset mining, but also undesirable for data analysis. For example, in marked basked analysis we want to avoid evaluation of association rules resulting from redundant itemsets since they describe the same set of transactions and therefore the same set of customers. Two common approaches to eliminate redundancies and to deal with the length problem are *maximal frequent itemsets* and *closed frequent itemsets*.

An itemset is maximal frequent if it has no superset that is frequent [Bay98]. In other words, one cannot add an item to a maximal frequent itemset with the result still being a frequent itemset. The set of maximal frequent itemsets implicitly and concisely represents the set of frequent itemsets, since any frequent itemset is a subset of a maximal frequent itemset. The number of maximal frequent itemsets is typically orders of magnitudes fewer than all frequent itemsets. However, extracting the set of frequent itemsets from the set of maximal frequent itemset is not straight forward. Maximal frequent itemsets are therefore not suitable for association rule generation.

Closed itemset mining was originally proposed in [PBTL99]. A closed itemset is an itemset having no proper subset with the same support, i.e., one cannot add an item to a closed itemset without changing the support of the itemset. Mining frequent closed itemsets also leads to orders of magnitudes smaller result sets than mining frequent itemsets. Moreover, the set of all frequent closed itemsets uniquely determines the set of all frequent itemsets, i.e., for each frequent itemset there exists exactly one closed frequent superset having the exact same support (regarding the set of transactions the itemset is contained in and not just the absolute support count). In Example 4-1 the set of closed frequent itemsets having support of at least 0.5 contains itemsets Z_2, Z_3, Z_4, Z_5, Z_7, Z_8, and Z_9. Itemset Z_1 is a subset of closed itemset Z_5 with exact same support and itemset Z_6 is a subset of closed itemset Z_9. The set of maximal frequent itemsets for the example in Example 4-1 only contains itemsets Z_8 and Z_9. However, we cannot directly derive a rule that describes the co-occurrence of items i_2 and i_3 in eight out of the ten transactions from the set of maximal frequent itemsets.

Contradiction patterns as defined in the following section are a special type of closed itemsets for relational databases. There exist various algorithms for efficient closed frequent itemset mining, such as A-CLOSE [PBTL99], CARPENTER [PCT+03], CHARM [ZH02], CLOSET+ [WHP03], FARMER [CTX+04], and MAFIA [BCG01]. Within this thesis we adopt two of these algorithms, namely CHARM (in this chapter) and CARPENTER (in Chapter 5) for mining contradiction patterns.

4.2 Patterns in Contradicting Databases

Contradiction patterns are a special kind of association rules. Using two databases r_1 and r_2 as shown in Figure 4-2, we illustrate characteristics in contradicting data to motivate our definition of contradiction patterns. Both relations follow the simple schema *R*(*ID*, *SPECIES*, *SEX*, *COLOR*, *SIZE*, *METHOD*). The relations represent information about certain properties for a set of amphibians analyzed independently by two laboratories or scientists. Each object has a unique identifier that forms the primary key of the relation. Also recorded is the classification of the species, the amphibian gender, color, and size. Attribute *METHOD* holds information about the experimental method used to determine values for the other attributes (except the primary key). The primary key attribute *ID* allows a simple identification of identical amphibians across the two databases. Tuples with matching primary key values from each of the databases are called *matching pairs*. Conflicts occur within matching pairs. They are highlighted by shaded cells in Figure 4-2.

For conflict resolution we aim to identify characteristics in the values regarding the occurrence of contradictions within each of the attributes (except the primary key attribute). We call these characteristics *patterns in contradicting data* (short *contradiction pattern*). These patterns help in providing answers to questions like "*Which are the conflict-causing attributes, values, or value pairs?*" and "*What kind of dependencies exists between the occurrences of contradictions in different attributes?*". A pattern is therefore a characteristic combination of values occurring with a certain frequency in conjunction with contradictions in a certain attribute. For a pattern to be of interest regarding the explanation of the contradictions it should not occur in combination with tuples that do not have a conflict in this exact attribute. Using the identified patterns, a domain expert can identify reasons for contradictions and specify proper actions for conflict resolution.

To highlight the usefulness of this concept we give examples. Consider again Figure 4-2. Obviously, there is a contradiction in each matching pair within attribute *SEX*. A good guess is that this high contradiction frequency results from different representations used for the gender of the amphibians by different research groups. In the first database values 'M' and 'W' are used to decode gender male and

r_1

ID	*SPECIES*	*SEX*	*COLOR*	*SIZE*	*METHOD*
1	Frog	F	Green	10.4	FScan++
2	Toad	F	Green	15.1	FScan++
3	Newt	M	Grey	16.7	AmpRd
4	Frog	F	Blue	9.8	FScan++
5	Toad	M	Blue	11.5	FScan++
6	Newt	M	Grey	19.6	AmpRd
7	Newt	F	Grey	17.2	AmpRd
8	Frog	M	Green	20.4	FScan++

r_2

ID	*SPECIES*	*SEX*	*COLOR*	*SIZE*	*METHOD*
1	Frog	1	Olive	10	FScan
2	Frog	1	Green	15	FScan
3	Newt	0	Greyish	16.7	AmpRd
4	Frog	1	Blue	10	FScan
5	Frog	0	Blue	12	FScan
6	Newt	0	Greyish	19.6	AmpRd
7	Newt	1	BlueGrey	12.7	AmpRd
8	Frog	0	Olive	20	FScan

Figure 4-2: Two overlapping databases containing information about gender, color, and size of a set of amphibians, analyzed independently by two laboratories or scientists. The experimental method used is given as well. Conflicts between the databases are highlighted by shaded cells.

female. The second database uses integers '0' and '1' instead. Regarding the characteristics concerning the conflicts in attribute *SIZE*, some inspection shows that there is a strong correlation with conflicts in the attribute denoting the experimental method used. The domain expert might conclude that one of the experimental methods makes errors in determining the size or at least is less precise than the other. On the other hand, the lonely conflict not being related to a methodical conflict appears to result from an arbitrary error, in this case probably a typographic error. Rare contradiction patterns like this one are likely to be left out by the pattern mining algorithm, depending on the parameter values used as described below. The conflicts in attributes *SPECIES* and *COLOR* are examples for imprecise data. Database r_1 further differs between frogs and toads for species classification while database r_2 does not differ between these subclasses. The same appears to be true for the representation of the color of newts where database r_2 uses a more fine graded representation for different grey colored newts.

4.2.1 Contradicting Databases

We now give a formal definition of contradiction patterns between pairs of overlapping databases. For simplicity, databases consist of a single relation r; they all follow the relational schema $R(A_1, \ldots, A_n)$. We assume the existence of a primary key constraint for schema R. Without loss of generality we assume A_1 to be the primary key attribute. We will use *ID* as synonym for attribute A_1. The primary key represents the unique object identifier for finding duplicate tuples between databases. Each database itself is therefore free of duplicates. Without loss of generality we assume $dom(ID) = \mathbb{N}$. We use $t\{j\}$ to refer to the tuple with primary key value $j, j \in \mathbb{N}$.

We start by defining pairs of matching tuples and conflicts between them. Later we give a relational representation for a set of matching tuples. This relational representation is used as input for our contradiction pattern mining algorithm. A pair of tuples from databases r_1 and r_2 is called a *matching pair* if they possess identical primary key values.

Definition 4.1: The *set of matching pairs* between databases r_1 and r_2 is denoted by $M(r_1, r_2)$, i.e., $M(r_1, r_2) = \{(t_1, t_2) \mid (t_1, t_2) \in r_1 \times r_2 \wedge t_1[ID] = t_2[ID]\}$. ■

Let $m = (t_1, t_2)$ be a matching pair from $M(r_1, r_2)$. The different tuples from m are denoted by $tup_1(m)$ and $tup_2(m)$, i.e., $tup_1(m) = t_1$ and $tup_2(m) = t_2$. The equal primary key value of both tuples is denoted by $id(m)$. A pair of databases r_1 and r_2 is called *overlapping* if $M(r_1, r_2) \neq \emptyset$. There might also be tuples in r_1 and r_2 without a matching partner in the other database. These tuples are called *unmatched*. Unmatched tuples cannot contain any conflicts (by our definition). Therefore, we only consider matched tuples for contradiction pattern mining.

Within a matching pair several conflicts may occur. The Boolean function $conflict(m, A)$ indicates whether contradicting values exists in the matching pair m for attribute A, i.e.,

$$conflict(m, A) = \begin{cases} true, \text{ if } tup_1(m)[A] \neq tup_2(m)[A] \\ false, \text{ else} \end{cases}.$$

A pair of databases r_1 and r_2 is called *contradicting*, if there is at least one conflict between them. We can define a relational representation of a set of matching pairs as a join of databases r_1 and r_2 on the primary key. For each attribute we add a conflict indicator. Each conflict indicator indicates whether a matching pair has a conflict in that particular attribute or not.

Definition 4.2: The *view of matching pairs*, denoted by $v(r_1, r_2)$, is a relational instance over schema $V(\underline{ID}, A_{21}, A_{22}, C_{A2}, A_{31}, A_{32}, C_{A3}, \ldots, A_{n1}, A_{n2}, C_{An})$. Attributes A_{i1} and A_{i2} are versions of attribute $A_i \in R$, $2 \leq i \leq n$. Attribute C_{Ai} is the *conflict indicator* for attribute $A_i \in R$. For each tuple $t \in v(r_1, r_2)$ there exists a corresponding matching pair $m \in M(r_1, r_2)$. For each attribute $B \in V$ the attribute value of t is defined as follows:

$$t[B] = \begin{cases} id(m), & \text{if } B = ID \\ tup_1(m)[A_i], & \text{if } B = A_{i1} \\ tup_2(m)[A_i], & \text{if } B = A_{i2} \\ conflict(m, A), & \text{if } B = C_A \end{cases}$$

■

Note that the primary key attribute occurs only once in the view of matching pairs and that there is no conflict indicator for that attribute. Figure 4-3 shows part of the view of matching pairs for the databases in Figure 4-2. For each conflict indicator, we further define two relations that represent disjoint subsets of $v(r_1, r_2)$ containing the conflicting and non-conflicting matching pairs for that respective conflict indicator.

Definition 4.3: The *set of conflicting matching pairs* for conflict indicator $C_A \in V$, denoted by $v_C(r_1, r_2, C_A)$, is the set of tuples $t \in v(r_1, r_2)$ having contradicting values for attribute A, i.e., $v_C(r_1, r_2, C_A) = \{t \mid t \in v(r_1, r_2) \wedge t[C_A]\}$. ■

Definition 4.4: The *set of non-conflicting matching pairs* for conflict indicator $C_A \in V$, denoted by $v_N(r_1, r_2, C_A)$, is the set of tuples $t \in v(r_1, r_2)$ not possessing a conflict in attribute A, i.e., $v_N(r_1, r_2, C_A) = \{t \mid t \in v(r_1, r_2) \wedge \neg t[C_A]\}$. ■

4.2.2 Contradiction Patterns

We now define terms and patterns over databases that form the basis for our definition of contradiction patterns.

Definition 4.5: A *term* τ over schema R is tuple (A, x), with attribute $A \in R$ and value $x \in dom(A)$. We also define $attr(\tau) = A$ and $value(\tau) = x$. ■

ID	*SPECIES*$_1$	*SPECIES*$_2$	$C_{SPECIES}$	*SEX*$_1$	*SEX*$_2$	C_{SEX}	*COLOR*$_1$	*COLOR*$_2$	C_{COLOR}
1	Frog	Frog	false	F	1	true	Green	Olive	true
2	Toad	Frog	true	F	1	true	Green	Green	false
3	Newt	Newt	false	M	0	true	Grey	Greyish	true
4	Frog	Frog	false	F	1	true	Blue	Blue	false
5	Toad	Frog	true	M	0	true	Blue	Blue	false
6	Newt	Newt	false	M	0	true	Grey	Greyish	true
7	Newt	Newt	false	F	1	true	Grey	BlueGrey	true
8	Frog	Frog	false	M	0	true	Green	Olive	true

Figure 4-3: Part of the view of matching pairs $v(r_1, r_2)$ for the databases r_1 and r_2 in Figure 4-2.

A term can be interpreted as a Boolean-function on tuples. A tuple t satisfies τ, denoted by $\tau(t) = true$, if $t[attr(\tau)] = value(\tau)$. By $\tau(r)$ we denote the set of tuples from r that satisfy τ. We say that the tuples in $\tau(r)$ are selected by τ.

Definition 4.6: A *pattern* ρ over schema R is a set of terms over schema R. For each attribute $A \in R$ there may only be one term $\tau \in \rho$, i.e., $\forall \tau_i, \tau_j \in \rho : attr(\tau_i) = attr(\tau_j) \Leftrightarrow \tau_i = \tau_j$. ■

A tuple t satisfies ρ, denoted by $\rho(t) = true$, if it satisfies each term within ρ. A pattern is therefore a conjunction of terms. The empty pattern is satisfied by each tuple of a database. Similar to the definitions above, $\rho(r)$ denotes the set of tuples satisfying ρ. We say that ρ selects the set of tuples $\rho(r)$ from the database r. We define the support a term or pattern has in a relation r as the fraction of tuples that satisfy them, i.e.,

$$sup(\tau, r) = \frac{|\tau(r)|}{|r|}, \text{ and } sup(\rho, r) = \frac{|\rho(r)|}{|r|}.$$

In accordance to the definition of closed itemsets in Section 4.1.3 we distinguish between patterns and closed patterns.

Definition 4.7: A pattern ρ with $\rho(r) \neq \{\}$ is a *closed pattern* for r if there does not exist a pattern $\rho' \supset \rho$ with $\rho'(r) = \rho(r)$, i.e., there exists no superset of ρ that selects the same set of tuples from r. ■

Note that a pattern that is closed for a database r_1 not necessarily is closed for a database r_2, i.e., the property of being a closed pattern can only be evaluated for a given database. An example pattern is ρ = {(*SPECIES*, 'Frog'), (*COLOR*, 'Green')}. This pattern is satisfied by tuples $t\{1\}$ and $t\{8\}$ in database r_1 and tuple $t\{2\}$ in database r_2 in Figure 4-2. The pattern is not closed in either of the databases. The corresponding closed pattern for database r_1 is ρ' = {(*SPECIES*, 'Frog'), (*COLOR*, 'Green'), (*METHOD*, 'FScan++')} and ρ'' = {(*SPECIES*, 'Frog'), (*COLOR*, 'Green'), (*METHOD*, 'FScan')} for database r_2.

Definition 4.8: We define the *corresponding closed pattern* for a pattern ρ in database r, denoted by $cp(\rho, r)$, as the closed pattern that selects exactly the same set of tuples from r than ρ, i.e., $\rho \subseteq cp(\rho, r)$ and $\rho(r) = cp(\rho, r)(r)$. ■

From the example it becomes apparent that the corresponding closed pattern for a pattern ρ is the set of terms that are common to all the tuples in $\rho(r)$. Based on the definition of patterns, we define contradiction patterns and their interestingness measures.

Definition 4.9: A *contradiction pattern* is a pattern that occurs in conjunction with conflicts between pairs of databases. Contradiction pattern are defined over schema V and are always interpreted in combination with a certain conflict indicator C_A. We denote contradiction patterns for conflict indicator C_A by ρ_A. ■

For example, the pattern ρ = {(*SPECIES*, 'Newt'), ($COLOR_1$, 'Grey'), (C_{COLOR}, 'true'), (*METHOD*, 'AmpRd')}[3] is a contradiction pattern for conflict indicator C_{COLOR} as it occurs in conjunction with

[3] For ease of presentation we group terms ($SPECIES_1$, 'Newt'), ($SPECIES_2$, 'Newt'), and ($C_{SPECIES}$, 'false') into a single term (*SPECIES*, 'Newt') denoting that there is no conflict in attribute *SPECIES* and that the value in both databases in 'Newt'. The same is done for the three terms concerning attribute *METHOD*.

three conflicts in attribute *COLOR* in the view of matching pairs shown in part in Figure 4-3. The pattern suggests the usage of different (e.g., more specific) representation of the color of grey newts in database r_2 and also shows that these differences are not related to the usage of different analysis methods. Pattern ρ is not considered a contradiction pattern for $C_{SPECIES}$ as it does not occur (for obvious reasons) in combination with conflicts in attribute *SPECIES*.

Interestingness Measures for Contradiction Patterns

Our definition of contradiction patterns considers any pattern that occurs in conjunction with a conflict a contradiction pattern. We define three measures of interest for contradiction patterns to filter those patterns that are most suitable to define characteristic conflict properties, i.e., represent a systematic conflict reason behind the occurrence of conflicts in a particular attribute.

Definition 4.10: The *conflict relevance* of a contradiction pattern ρ_A is defined as the fraction of matching pairs having a conflict in attribute A that satisfy ρ_A, i.e., the support $sup(\rho_A, v_C(r_1, r_2, C_A))$ of ρ_A in $v_C(r_1, r_2, C_A)$, denoted by $rel(\rho_A, r_1, r_2, C_A)$. ■

Conflict relevance is a measure for the support (or relevance) of a pattern ρ_A in the set of matching pairs having a conflict in a certain attribute. The higher the conflict relevance the more conflicts are captured by the pattern. Since contradiction patterns define groups of conflicts assumed to result from the same conflict reason, we are particularly interested in patterns having high conflict relevance.

Definition 4.11: The *conflict potential* of a contradiction pattern ρ_A is defined as the ratio of matching pairs in $v_C(r_1, r_2, C_A)$ that satisfy ρ_A over the total number of matching pairs that satisfy ρ_A, i.e.,

$$pot(\rho_A, r_1, r_2, C_A) = \frac{|\rho_A(v_C(r_1, r_2, C_A))|}{|\rho_A(v(r_1, r_2))|}.$$

■

Conflict potential is a measure for the accuracy of a contradiction pattern in 'predicting' a conflict in attribute A. The higher the conflict potential of a pattern ρ_A, the higher the probability for a matching pair that satisfies pattern ρ_A to also possess a conflict in attribute A. The occurrence of contradiction patterns having high conflict potential is closely related to the occurrence of conflicts in the particular attribute. On the other hand, patterns having low conflict potential are almost meaningless in our scenario since their occurrence is unrelated to the occurrence of conflicts. These patterns point only by low chance towards systematic conflict reasons.

Our third interestingness measure for contradiction patterns is motivated by the work on association rule mining in data sets having skewed support distribution [XTK03]. Association rule mining algorithms using a support-based pruning strategy tend to generate uninteresting patterns involving items with substantially different support levels from databases with skewed support distribution. For instance, in a transactional database the frequency of items like milk, bread, and butter is expected to significantly higher than the frequency of luxury goods like caviar. Therefore, it is not surprising to find milk present in transactions that contain caviar for example. Thus, patterns involving items with substantially different support levels tend to be uninteresting for analytic purposes.

It is important to note that for a contradiction pattern, we only want to include terms that are relevant or interesting in conjunction with the occurrence of a certain type of contradiction. This means that we

do not want to combine terms with largely differing conflict relevance, as their combination might not yield any important information for the domain expert evaluating the computed contradiction patterns afterwards. For example, occurrence of term (C_{SEX}, 'true') in contradiction pattern ρ_{COLOR} = {(*SPECIES*, 'Newt'), (C_{SEX}, 'true'), (*COLOR*$_1$, 'Grey'), (C_{COLOR}, 'true'), (*METHOD*, 'AmpRd')} is not surprising since each of the matching pairs in the databases in Figure 4-2 posses a conflict in attribute *SEX*. We use a relevance deviation threshold to avoid such uninteresting terms in contradiction patterns.

Definition 4.12: The *relevance deviation threshold* for a pattern ρ is defined following the definition in [XTK03] as:

$$reldev(\rho, r_1, r_2, A) = 1 - \frac{min(rel(\{\tau\}, r_1, r_2, C_A)}{max(rel(\{\tau\}, r_1, r_2, C_A)}, \forall \tau \in \rho \, .$$

■

Problem Statement

Based on the above definition, the problem of contradiction pattern mining is defined as follows: Given a pair of databases r_1 and r_2 find the set of contradiction patterns for $v(r_1, r_2)$ having conflict potential and conflict relevance above given thresholds and relevance deviation below a given threshold.

4.3 Mining Contradiction Patterns

We now describe our algorithm for mining contradiction patterns. We start by showing the relationship between contradiction patterns and a special type of association rules, called class association rules. We then describe our adoption of existing association rule mining techniques for contradiction pattern mining. In general, association rules over relational databases can be defined using patterns and terms instead of itemsets and items. An association rule $\rho_1 \Rightarrow \rho_2$ in a database r is a statement about the co-occurrence of patterns ρ_1 and ρ_2 in tuples of r, i.e., tuples that satisfy ρ_1 also satisfy ρ_2 with a probability of $conf(\rho_1 \Rightarrow \rho_2)$. Support and confidence for association rules $\rho_1 \Rightarrow \rho_2$ are defined similar to association rules over transactional databases, i.e.,

$$sup(\rho_1 \Rightarrow \rho_2) = \frac{|\rho_1(r) \cap \rho_2(r)\}|}{|r|}, \text{ and } conf(\rho_1 \Rightarrow \rho_2) = \frac{|\rho_1(r) \cap \rho_2(r)|}{|\rho_1(r)|}.$$

Contradiction patterns are closely related to class association rules defined in [LHM98]. A class association rule is an association rule having a right hand side that contains a single item. The item in the consequent of a class association rule is an element from a set of possible class labels. The set of class association rules for each class label is used to build a classifier for the transactions that satisfy the rules. Accordingly, a conflict indicator is a class label '*Has conflict in attribute A*' and a contradiction pattern ρ_A can be interpreted as a class association rule $\rho_A \Rightarrow \{(C_A, \text{'true'})\}$ for $v(r_1, r_2)$. In the following we show the relatedness of conflict potential and conflict relevance for ρ_A with confidence and support for association rules $\rho_A \Rightarrow \{(C_A, \text{'true'})\}$ and $\{(C_A, \text{'true'})\} \Rightarrow \rho_A$. We regard a contradiction pattern as a special association rule $\rho_A \Leftrightarrow \{(C_A, \text{'true'})\}$, being a probabilistic statement describing the co-occurrence of a pattern ρ_A with conflicts in attribute A, i.e., (i) a matching pair that satisfy ρ_A has a

conflict in A with probability $conf(\rho_A \Rightarrow \{(C_A, \text{'true'})\})$, and (ii) a matching pair having a conflict in A satisfies ρ_A with probability $conf(\{(C_A, \text{'true'})\} \Rightarrow \rho_A)$. We use ρ_C to denote pattern $\{(C_A, \text{'true'})\}$ and v and v_C as abbreviation for $v(r_1, r_2)$ and $v_C(r_1, r_2, C_A)$ respectively.

Following our definitions, $\rho_C(v)$ denotes the set of tuples from v that satisfy term $(C_A, \text{'true'})$. According to Definition 4.3 this set equals v_C. Therefore, it holds that $\rho_A(v) \cap \rho_C(v) = \rho_A(v_C)$. We use this equality to show that support for $\rho_A \Rightarrow \rho_C$ is related to conflict relevance $rel(\rho_A, r_1, r_2, C_A)$ while confidence for $\rho_A \Rightarrow \rho_C$ equals conflict potential $pot(\rho_A, r_1, r_2, C_A)$:

$$sup(\rho_A \Rightarrow \rho_C) = \frac{|\rho_A(v) \cap \rho_C(v)|}{|v|} = \frac{|v_C|}{|v_C|} \cdot \frac{|\rho_A(v_C)|}{|v|} = \frac{|v_C|}{|v|} \cdot rel(\rho_A, r_1, r_2, C_A), \text{ and}$$

$$conf(\rho_A \Rightarrow \rho_C) = \frac{|\rho_A(v) \cap \rho_C(v)\}|}{|\rho_A(v)|} = \frac{|\rho_A(v_C)|}{|\rho_A(v)|} = pot(\rho_A, r_1, r_2, C_A).$$

Accordingly, we show that support and confidences for $\rho_C \Rightarrow \rho_A$ are also related to conflict relevance and conflict potential:

$$sup(\rho_C \Rightarrow \rho_A) = sup(\rho_A \Rightarrow \rho_C) = \frac{|v_C|}{|v|} \cdot rel(\rho_A, r_1, r_2, C_A), \text{ and}$$

$$conf(\rho_C \Rightarrow \rho_A) = \frac{|\rho_A(v) \cap \rho_C(v)|}{|\rho_C(v)|} = \frac{|\rho_A(v_C)|}{|v_C|} = rel(\rho_A, r_1, r_2, C_A).$$

These equations show the relationship of contradiction pattern mining using conflict potential and conflict relevance threshold to class association rule mining using support and confidence thresholds. In [LHM98] the authors present CBA-RG, an algorithm for mining class association rules for a database r where each tuple contains an additional class label. The algorithm is a variation of APRIORI. However, the following problems make adopting CBA-RG infeasible for contradiction pattern mining:

1. We discussed in Section 4.1.3 that association rule mining algorithms normally generate huge amounts of redundant rules. Using CBA-RG for contradiction pattern mining will therefore overwhelm the expert user with a large amount of redundant patterns. Furthermore, mining the complete set of patterns is inefficient or even infeasible for large databases.
2. CBA-RG used support and confidence for pattern pruning. In contradiction pattern mining, we additionally consider a third interestingness measure relevance deviation.
3. Using the value of $t[C_A]$ as class label will also generate class association rules for class label '$t[C_A]$ = '*false*'', i.e., for those cases where there does not exist a conflict in a matching pair. However, in contradiction pattern mining, we are only interested in rules that are related to the occurrence of conflicts.

Our initial work on contradiction pattern mining was based on an APRIORI-like approach. However, the huge amount of patterns returned by the initial algorithm lead to the development of an advanced mining algorithm based on closed frequent itemset mining. The set of closed patterns can be orders of magnitude smaller than the set of patterns while containing all the information that is needed by the expert user in his/her task to identify possible reasons for systematic differences. Therefore, we adopt closed frequent itemset mining algorithms for contradiction pattern mining.

4.3.1 CPMine – Contradiction Pattern Mining Algorithm

We now describe our contradiction pattern mining algorithm CPMINE. The algorithm is an adoption of the closed frequent itemset mining algorithm CHARM [ZH02]. CHARM uses a depth-first approach for pattern enumeration and support counting is done by intersecting tid-lists. Frequent itemset mining using CHARM is based on candidate generation. We choose not to use an FP-tree approach for closed itemset mining as it is employed for example by CLOSET+ [WHP03]. First, in our setting all transactions, i.e. tuples, are of the same size. Therefore, we expect tuples to not share long prefixes. The prefix tree data structure for representing transaction data is assumed to result is a very flat tree having a large fan-out at the root level. Second, the patterns we are generating cannot have more terms than attributes in the view of matching pairs. The levels of candidate generation are limited by this number of attributes. Further restrictions on the combinability of terms belonging to the same attribute benefits candidate generation since they reduce the number of possible candidates. Third, pruning based on relevance deviation is performed in a more efficient way when using a candidate generation-based approach.

CHARM uses a tree structure, called IT-tree, for enumerating closed itemsets. An example IT-tree is outlined in Figure 4-4. Each node of the tree is a *prefix-list pair*. The prefix (shown in curly brackets) describes an itemset and the list contains all the frequent items the prefix can be extended with. The root of the tree has an empty prefix and the elements in the list are ordered by a given sorting criteria, e.g. in lexicographical order, by their support, etc. For contradiction pattern mining the prefix in the nodes of the IT-tree and the items in the extension list are patterns instead of itemsets and items.

CPMINE (shown in Figure 4-5) takes two databases r_1 and r_2, a conflict indicator C_A, and conflict potential, conflict relevance, and relevance deviation thresholds as parameters. The algorithm returns the set of contradiction patterns for attribute A that satisfy the given thresholds. CPMINE first determines the set of frequent terms from the view of matching pairs for r_1 and r_2 (*line 3*). This set is generated in a single pass over the data. Each distinct attribute-value pair represents a term for which the support in v_C is counted. Terms not having sufficient conflict relevance are pruned. The conflict indicator C_A is excluded in term enumeration. For each frequent term a pattern of length *1* is generated. This pattern set forms the extension list for the root node of the IT-tree. The prefix pattern of the root node is empty. The main computation is performed in subroutine *CPMine-Extend* (*line 4*). The last step of CPMINE removes patterns returned by *CPMine-Extend* having conflict potential below min_{pot}.

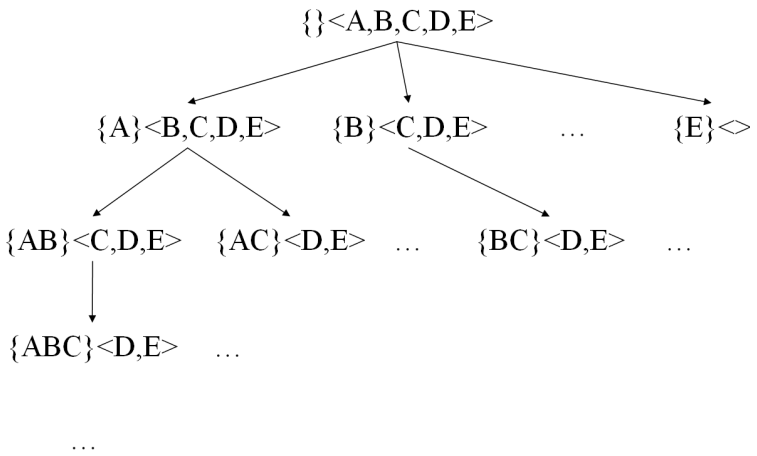


Figure 4-4: Each node in the IT-tree has a prefix itemset and a list of possible extensions.

```
1 CPMine(r1, r2, CA, minrel, minpot, maxdev) {
2   P := {};
3   root := ({}, frequentTerms(r1, r2, CA, minrel));
4   CPMine-Extend(root, root, P, r1, r2, CA, minrel, maxdev);
5   return removePatterns(P, minpot);
6 }
```

Figure 4-5: CPMINE algorithm for mining contradiction patterns for an attribute *A*.

CPMine-Extend (shown in Figure 4-6) recursively builds the IT-tree in depth-first order. For each element in the extension list of the current node n_{curr} a child node n_{child} is added to the tree (*line 3*). We then generate the extension list for the newly added node (*line 4-11*). The extension list contains all patterns from the parent's extension list that appear later in order '$>_\rho$' than the prefix of the newly generated node. The authors of CHARM mention that different ordering criteria are possible for the elements in the extension list. We use the support of a pattern in increasing order as our sorting criteria, i.e., $sup(\rho_i, v) < sup(\rho_j, v)$ implies $\rho_i >_\rho \rho_j$ and vice versa. This definition of pattern order brings patterns with equal support together and maximizes the occurrence of properties 1 and 2 in *CHARM-Property* (see below). For each node to be added to the extension list of n_{child} we check whether it is compatible with ρ_i (*line 5*). Two patterns ρ_i and ρ_j are considered incompatible if:

a) they contain terms $\tau_i \in \rho_i$, $\tau_j \in \rho_j$ for the same attribute but different values, i.e., $attr(\tau_i) = attr(\tau_j) \wedge value(\tau_i) \neq value(\tau_j)$, or

b) the relevance deviation of their terms is above the given threshold.

```
1 CPMine-Extend(root, ncurr, P, r1, r2, CA, minrel, maxdev) {
2   for each ρi ∈ list(ncurr) {
3     nchild := addChild(ncurr, (ρi, {}));
4     for each ρj ∈ list(ncurr) with ρj >ρ ρi {
5       if (compatible(ρi, ρj, maxdev)) {
6         ρm := ρi ∪ ρj;
7         if (rel(ρm, r1, r2, CA) > minrel) {
8           CHARM-Property(ρi, ρj, ρm, root, ncurr, nchild);
9         }
10      }
11    }
12    if (list(nchild) ≠ {}) {
13      CPMine-Extend(root, nchild, P, r1, r2, CA, minrel, maxdev);
14    }
15    removeChild(ncurr, nchild);
16    if (¬subsumed(prefix(nchild), P)) {
17      P := P ∪ prefix(nchild);
18    }
19  }
20}
```

Figure 4-6: *CPMine-Extend* recursively extends the IT-tree and enumerates the complete set of contradiction patterns that satisfy the conflict relevance and relevance deviation thresholds.

Note that due to point b) the patterns returned by CPMINE are not necessarily closed as we omit terms that violate the relevance deviation threshold. However, we still avoid most of the redundancies in the result set by exploiting the properties of closed patterns as described below. Patterns that are compatible are merged (*line 6*) and the conflict relevance for the result is determined. We call function *CHARM-Property* if the merged pattern ρ_m has sufficient support in v_C, i.e., sufficient conflict relevance (*lines 7-9*).

In *CHARM-Property* nodes in the tree may be modified by replacing or removing patterns in their prefix pattern as well as in the elements of their extension lists. CHARM leverages two basic properties of patterns for efficient closed pattern enumeration. Given two patterns ρ_1 and ρ_2 it holds that:

1. If $\rho_1(r) = \rho_2(r)$, then $cp(\rho_1, r) = cp(\rho_2, r) = cp(\rho_1 \cup \rho_2, r)$, i.e., two patterns that select the same set of tuples have the same corresponding closed pattern that is equal to the corresponding closed pattern of the union of their terms. Recall that $cp(\rho, r)$ is defined as the set of terms common to all tuples in $\rho(r)$ (Definition 4.8). Two patterns that select the same set of tuples clearly have the same corresponding closed pattern and that closed pattern has to contain at least all terms that are contained in either of the patterns.
2. If $\rho_1(r) \subset \rho_2(r)$, then $cp(\rho_1, r) = cp(\rho_1 \cup \rho_2, r) \neq cp(\rho_2, r)$, i.e., a pattern ρ_1 that selects a subset of the tuples selected by a pattern ρ_2 has the same corresponding closed pattern than the union of their terms. However, the corresponding closed pattern for ρ_2 is different since it selects a different set of tuples.

The first property implies that we replace patterns ρ_1 and ρ_2 with $\rho_1 \cup \rho_2$. The second property implies that we only replace ρ_1 with $\rho_1 \cup \rho_2$ while retaining ρ_2 as a separate pattern. Whenever we replace or remove a pattern we traverse the tree from the root to the leaf nodes and replace/delete each occurrence of the pattern in the prefix pattern of the node as well as in the extension lists. Based on these properties we distinguish four cases in *CHARM-Property*:

1. $\rho_i(v) = \rho_j(v)$: We remove ρ_j from the extension list of node n_{curr} and replace every occurrence of ρ_i in the IT-tree with ρ_m.
2. $\rho_i(v) \subset \rho_j(v)$: We replace every occurrence of ρ_i with ρ_m.
3. $\rho_i(v) \supset \rho_j(v)$: We remove ρ_j from the extension list of node n_{curr} and continue with ρ_m instead by adding it the extension list of node n_{child}.
4. $\rho_i(v) \not\subset \rho_j(v) \wedge \rho_j(v) \not\subset \rho_i(v)$: We add ρ_m to the extension list of node n_{child}.

Clearly, the first property is the most desirable one regarding reduction of the number of patterns in the IT-tree. We allow for this by sorting the patterns according to their support. In addition, pruning based on relevance deviation eliminates many of the other three cases. After generating the extension list of node n_{child}, we call *CPMine-Extend* recursively if the extension list is not empty (*line 12-14*). Once all children of n_{child} have been processed its prefix is added to the result set of contradiction patterns. Note that the prefix may have changed due to replacements performed in *CHARM-Property*. We have to check whether the prefix is subsumed by existing patterns in the result set that have been added while traversing the children of n_{child} in recursive calls the *CPMine-Extend*. In that case the prefix of n_{child} will not be added to the result set.

CHARM maintains tid-lists for items to perform fast checking of itemset support. In CPMINE we use disjunctive lists $tidlist_C$ and $tidlist_N$ that represent the primary keys of tuples in v_C and v_N satisfying the pattern or term, respectively. The tid-list of a pattern is the intersection of the according tid-lists of all of the terms in the pattern. The primary keys in a tid-list are sorted in ascending order to enable efficient intersection. Conflict potential and conflict relevance for candidate patterns can be determined using the size of tid-list, i.e., $|tidlist_C(\rho_A)| = |\rho_A(v_C)|$, $|tidlist_N(\rho_A)| = |\rho_A(v_C)|$, and $|tidlist_C(\rho_A)| + |tidlist_N(\rho_A)| = |\rho_A(v)|$.

4.4 Experimental Results

For our experiments, we used two relational instances of protein structure data that were used for the FIRST GERMAN INFORMATION QUALITY CONTEST in 2004 [MMN05]. The first relation is directly derived from PDB flat-files using the parser from the BIOPYTHON PROJECT (www.biopython.org). The second relation results from parsing MMCIF-files using the OPENMMS TOOLKIT. We refer to these datasets as PDB and OPENMMS respectively. The schema has a total of 10 attributes containing information about the deposition date and year of an entry, the resolution of the protein structure described as well as the experimental method used for resolution determination. Table 4-1 lists the attributes and statistics about (i) the number of conflicts occurring within the attribute, (ii) the number of distinct values in PDB, (iii) the number of distinct values in OPENMMS, and (iv) the total number of distinct values in both sources. The relational instance resulting from the PDB flat-files has 26,764 tuples. The instance resulting from OPENMMS contains 24,202 tuples. Identification of matching tuples is trivial using the original *PDB_ID* for all entries that is also an attribute of relation *PDB_ENTRY*. The number of matching pairs between PDB and OPENMMS is 23,614.

Table 4-1: Statistics about the attribute and their values in the view of matching pairs resulting from PDB and OPENMMS. We use A1-A10 to denote the attributes in the following tables and figures.

Attribute	Conflicts between PDB and OpenMMS	Distinct Values in PDB	Distinct Values in OpenMMS	Total Number of Distinct Values
(A1) DEPOSITION_YEAR	66	34	34	37
(A2) DEPOSITION_DATE	112	4,252	4,083	4,328
(A3) RELEASE_DATE	11,028	1,612	1,255	1,737
(A4) AUTHORS	22,195	15,839	22,263	37,045
(A5) STRUCTURE_METHOD	2,853	39	101	138
(A6) RESOLUTION	9,755	322	376	419
(A7) R_FREE	23,604	1	1,796	1,797
(A8) REFINEMENT_METHOD	23,614	2	500	502
(A9) NMR_STRUCTURES	2,355	59	20	63
(A10) CHAINS	2,381	35	35	40

Parameter Values

Table 4-2 lists the number of contradiction patterns for each attribute when using different values for these parameters. In general, the number of patterns decreases with increasing conflict relevance thresholds. For small conflict relevance thresholds the relevance deviation parameter has a significant influence on the number of returned patterns. On the other hand, influence of conflict potential is limited. For attribute *REFINEMENT_METHOD* the conflict potential threshold has no influence at all since every term has a potential of *1* (the set of non-conflicting matching pairs is empty for this attribute). Influence of relevance deviation decreases with increasing conflict relevance thresholds as the highly deviating terms are already filtered by the relevance threshold. Therefore, with increasing conflict relevance the conflict potential becomes the only other significant parameter (for attributes with modest conflict rates).

Table 4-2: The number of contradiction patterns for each attribute is shown for different conflict relevance (min_{rel}), conflict potential (min_{pot}), and relevance deviation (min_{dev}) thresholds.

min_{rel}	min_{pot}	min_{dev}	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
2	25	25	51	42	758	480	510	445	476	763	558	39
2	25	75	473	411	7,639	8,163	2,651	12,241	10,844	15,369	2,701	2,148
2	75	25	23	30	30	480	326	199	476	763	290	27
2	75	75	241	351	2,818	8,163	1,903	6,545	10,844	15,369	1,461	1,510
50	25	25	20	1	602	288	488	126	288	574	540	1
50	25	75	32	8	1,320	578	720	667	575	1,150	910	3
50	75	25	1	1	29	288	312	18	288	574	274	1
50	75	75	11	8	304	578	508	42	575	1,150	457	3
75	25	25	10	0	480	286	183	42	248	496	461	0
75	25	75	10	0	480	286	183	42	248	496	461	0
75	75	25	0	0	0	286	90	4	248	496	232	0
75	75	75	0	0	0	286	90	4	248	496	232	0

Figure 4-7-Figure 4-10 show the influence that each mining parameter has on the total number of patterns found. In these experiments, we choose fixed (and weak constraining) values for two parameters and vary the third parameter. For each attribute the number of patterns is shown. In Figure 4-7 the influence of the relevance deviation is shown for conflict relevance of 2% and conflict potential 25%. For threshold values between 0% (no deviation allowed at all) and 40% the number of patterns remains almost constant at an overall low level. The number of patterns starts to increase significantly for threshold values that are above 50%. A relevance deviation threshold of 100% means that the parameter is not considered at all and the number of patterns gets huge for attributes with many conflicts. Note that this behavior is opposite to the other parameters where higher values restrict the patterns more. Figure 4-8 shows basically the same properties for the conflict relevance parameter using a conflict potential threshold of 25% and relevance deviation at 100%. For our experiment, we used conflict relevance thresholds between 100% and 10%. Note that the maximal number of patterns is smaller than in the previous experiment due to the increased conflict potential threshold.

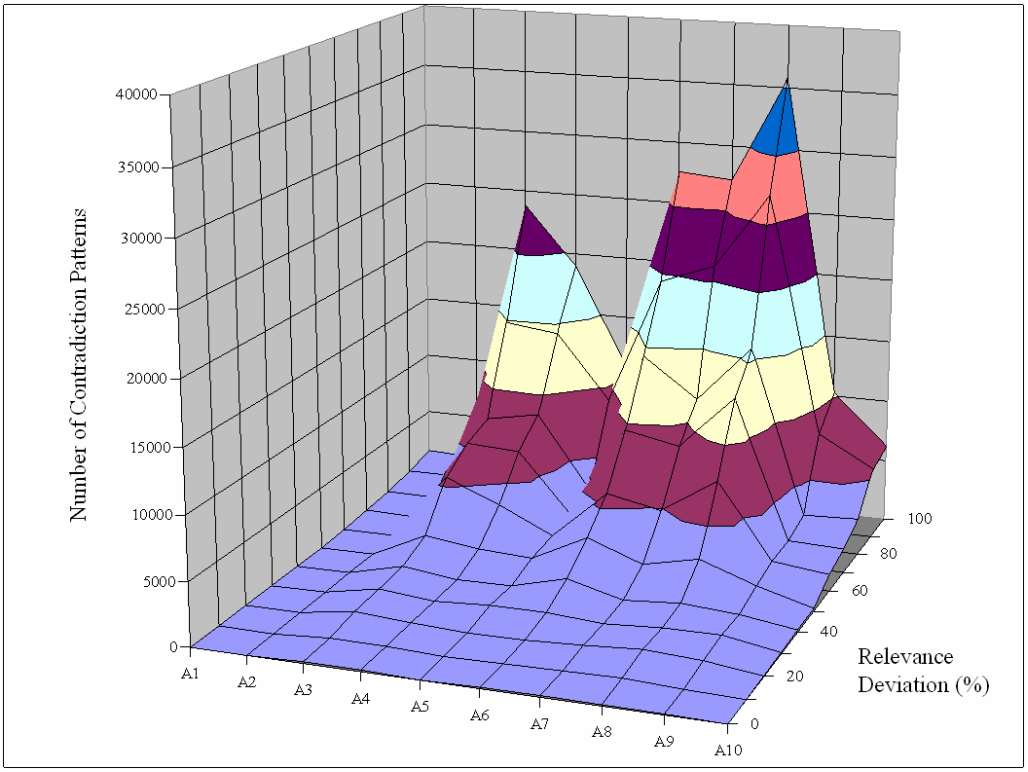


Figure 4-7: Influence of relevance deviation in contradiction pattern mining using fixed thresholds for conflict relevance (2%) and conflict potential (25%). Attributes are on the x-axis, the number of patterns is shown on the y-axis, and the z-axis shows the relevance deviation between 0% and 100%.

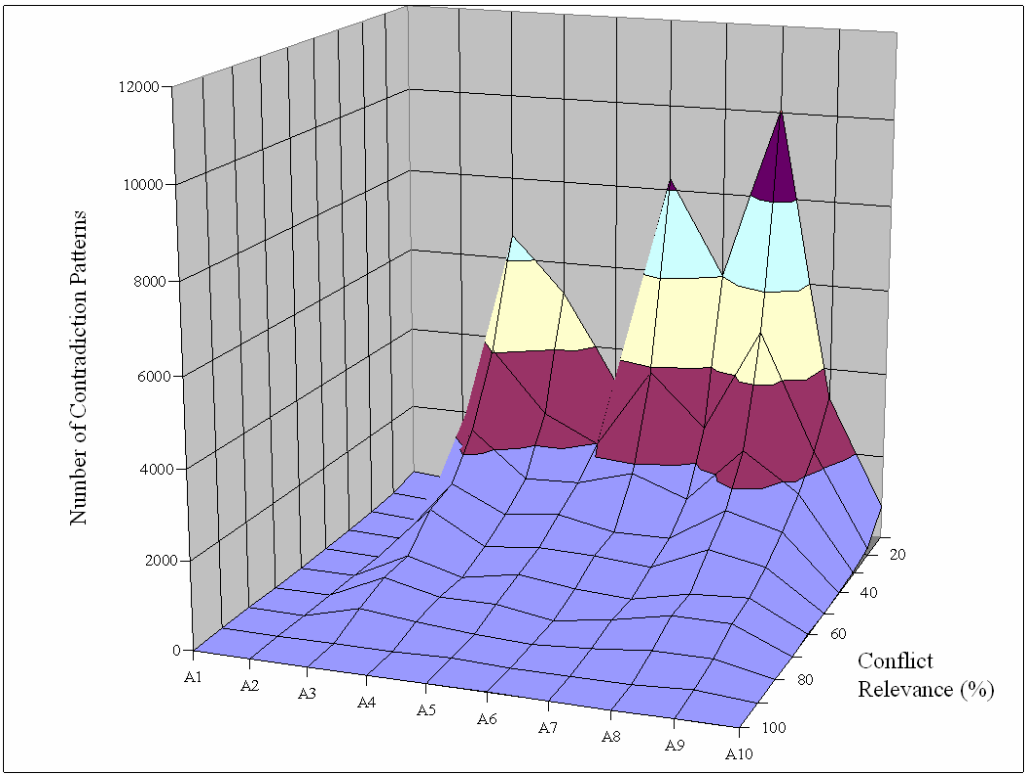


Figure 4-8: Influence of conflict relevance in contradiction pattern mining using fixed thresholds for conflict potential (25%) and relevance deviation (100%). The z-axis shows the conflict relevance starting at 10%.

For attributes with many conflicts the number of patterns increases significantly when lowering the conflict relevance threshold. For attributes with small number of conflicts the overall number of terms occurring in conjunction with these conflicts in general is small and so is the number of patterns. The inability of conflict potential to constrain the number of patterns for attributes containing a large number of conflicts is shown in Figure 4-9. For attribute *REFINEMENT_METHOD* the number remains constant and *R_FREE* also shows only a small variation. For all other attributes, increasing the conflict potential threshold leads to a decrease in the number of contradiction patterns. However, the influence is not as obvious as that of the relevance deviation parameter for example. To justify our definition of the conflict potential, we performed experiments with varying conflict relevance and conflict potential thresholds to find out how the number of patterns for individual attributes changes. Figure 4-10 depicts the numbers of patterns for attribute *STRUCTURE_METHOD* when varying the parameters between 100% and 10%. Attribute *STRUCTURE_METHOD* has a conflict frequency of about 10%, i.e. 2,853 of 23,614 matching pairs show a conflict in this attribute. The relevance deviation was fixed at 50% for the experiment. Compared to Table 4-2 and Figure 4-9, influence of the conflict potential parameter becomes clear in Figure 4-10. Especially for small conflict relevance values the potential significantly constraints the number of patterns. This behavior is desirable when searching for patterns that describe smaller subsets of conflicts in an attribute with high accuracy. Only in a few rare cases we expect to be able to find a single pattern that describes all the conflicts in an attribute. Instead, we are often looking for patterns that are closely related to the occurrence of conflicts. The number of patterns that are in such close relationship is significantly lower that the number of patterns fulfilling the other two thresholds alone. For small conflict relevance thresholds a conflict potential of 80% appears to be a good discriminator (for attribute *STRUCTURE_METHOD*).

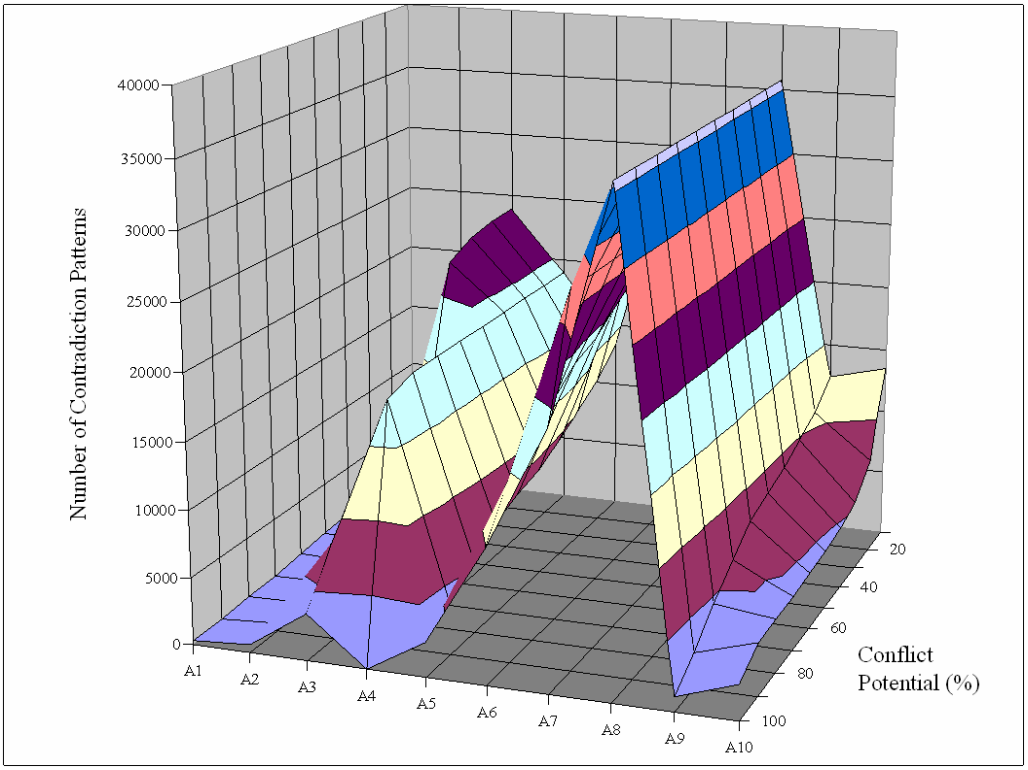


Figure 4-9: Influence of conflict potential on the number of contradiction patterns. We use a fixed conflict relevance threshold of 2% and fixed relevance deviation threshold of 100%. The z-axis shows the varying conflict potential values.

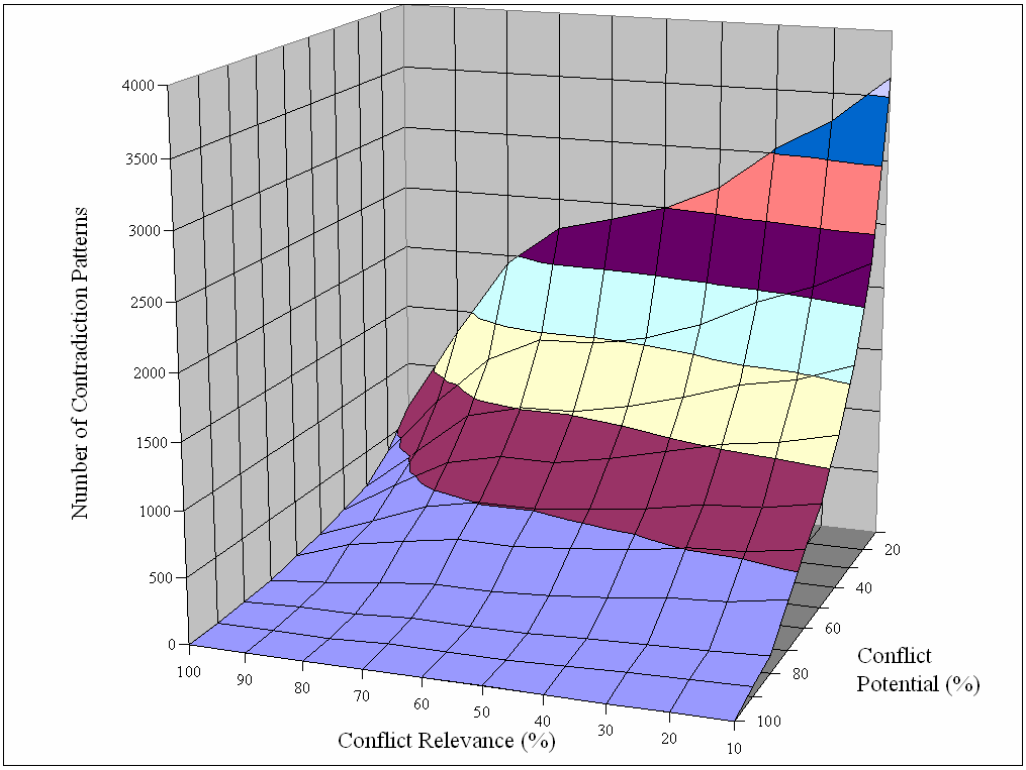


Figure 4-10: Distribution of the number of contradiction patterns returned by CPMINE over different values for conflict potential and conflict relevance for attribute *STRUCTURE_METHOD*. Especially for small conflict relevance thresholds conflict potential is significant. However, the influence decreases as the number of conflicts in the attribute increases as terms have higher potential initially.

Pattern Evaluation

In the following we discuss example contradiction patterns generated by our various experiments by giving possible interpretation for them:

ρ_{R_FREE}: {(*R_FREE*$_{PDB}$, '+0.00000E+000')}

The numbers in Table 4-1 indicate that there is only a single distinct value in attribute *R_FREE* in the PDB dataset. Looking at the data reveals that the value is '+0.00000E+000'. Accordingly, there is a contradiction pattern revealing that this value occurs in all conflicts. Since the value is equal to zero it is expected that the reason for the conflicts in attribute *R_FREE* are a consequence of the flat-file parser not considering the value appropriately. The same is true for conflicts in attribute *REFINEMENT_PROGRAM* where for the great majority of conflicts the value in PDB is 'unknown'.

$\rho_{DEPOSITION\_YEAR}$: {(*DEPOSITION_YEAR*$_{PDB}$, '2000'), (*DEPOSITION_YEAR*$_{OPENMMS}$, '1900')}

Each time the value '1900' occurs in attribute *DEPOSITION_YEAR* in OPENMMS the according value in PDB is '2000'. This pattern has a conflict relevance of about 50%. Conflict potential and relevance deviation are 100% and 0% respectively. Deposition years are extracted from deposition dates of PDB entry. Looking at the PDB flat-file format reveals that the conflicts presumably results from parsing errors in conjunction with the Y2K problem. The deposition date is represented in format DD-MM-YY in flat files. Thus, in some cases a year '00' is transformed into year '1900' by the parser. Note that this is not true for all deposition dates having '00' as their year value. However, knowing that PDB was established in 1970s the solution for these conflicts is obvious.

$\rho_{\text{DEPOSITION\_YEAR}}$: {($C_{DEPOSITION\_DATE}$, 'true')}

This follows directly as *DEPOSITION_YEAR* is the year fraction of *DEPOSITION_DATE*. Having determined this, we can conclude that the conflicts in *DEPOSITION_YEAR* can be ignored since the problem must be resolved in the *DEPOSITION_DATE*.

$\rho_{\text{RESOLUTION}}$: {($STRUCTURE\_METHOD_{OPENMMS}$, 'NMR')}

When taking a closer look at the values involved in attributes *RESOLUTION* and *STRUCTURE_METHOD* we see that neither sources stores meaningful values for *RESOLUTION* if *STRUCTURE_METHOD* has value 'NMR'. However, the sources represent this fact differently: PDB uses the value '0.0', while OPENMMS uses NULL.

$\rho_{\text{NMR\_STRUCTURES}}$: {($STRUCTURE\_METHOD_{PDB}$, 'NMR') ($STRUCTURE\_METHOD_{OPENMMS}$, 'NMR, n STRUCTURES'), ($NMR\_STRUCTURES_{PDB}$, 'n')} where n takes different values like 10 or 20 for example

This pattern reveals that in OPENMMS for structure method NMR the number of identified structures is encoded in the method name while PDB separately lists the number of structures in attribute *NMR_STRUCTURES*. The first term is not included in these patterns for small relevance deviation values due to its occurrence with many of these (syntactical) different conflicts.

We also find patterns for conflicts in attribute *CHAINS* where for a particular list of authors ('PELLETIER, H., SAWAYA, M.R.') using a program for structure refinement ('TNT 5-D') the number of chains varies between the tuples in PDB ('3') and OPENMMS ('1'). We were not able to give a reasonable explanation for this pattern. However, the pattern highlights conflict peculiarities that require further investigation and may lead to valuable information on the actual conflict reasons.

4.5 Summary and Related Work

We presented an algorithm for finding patterns of contradictions in semantically overlapping, yet different datasets. Whenever such datasets are merged into a uniform database with "a single truth", conflicts need to be identified and resolved which usually requires costly expert inspection. Our algorithm helps in that it points the expert's attention to the most prevalent patterns of conflicts. The patters that we find are special cases of association rules. We adopt an existing algorithm for mining closed frequent itemsets for mining contradictory data. We define a measure of interestingness for our patterns consisting of three parameters and discuss in our experimental section practical parameter values. The areas of related work to our approach are assessment of data quality (regarding accuracy and uniformity), statistical comparison of data sets, and association rule mining. The latter has been discussed Section 4.1. Data quality and assessing scores for the quality of data sources has been discussed in Chapter 2.

The comparison of data sets using statistical methods is described in [BP01, WBN03]. Other than in our approach, the authors do not compare overlapping data sources representing the same real-world objects. They are also not interested in the actual values causing the differences between the data sets. Using set comparison techniques the authors try to identify trends or other noticeable problems. For example, comparing customer data from different branches can reveal customer preferences and behavior by region, by group, or by month. However, the techniques used are comparable to those that

we use in this chapter. In [BP01] the authors present STUCCO, an algorithm for mining contrast sets. A contrast set is a set of attribute-value pairs that differ meaningful in their distribution in different datasets. Contradiction pattern mining can be seen as a special case of contrast set mining when considering the set of conflicting matching pairs (v_C) and non-conflicting matching pairs (v_N) for an attribute as different sets. Contradiction patterns are patterns whose distribution differs significantly between these two sets. The algorithm presented in [BP01] is designed to identify sets of attribute-value pairs that show any kind of difference in their distribution within the two sets. For contradiction patterns we are mainly interested in patterns that have great difference in their distribution between two databases. In [WBN03] contrast set mining is compared with decision tree induction and class association rule mining to describe differences between two datasets. In a study with retail collaborators the patterns found by each of the methods were assessed regarding their potential usefulness. The results for two different retailer datasets show that class association rule mining has the overall best ability to identify potentially surprising and useful patterns. Our approach is closely related to class association rule mining. However, we use a closed pattern mining approach to reduce redundancies in the set of contradiction patterns and define an additional measure of interestingness to help reduce the number of patterns and focus on those patterns only containing terms that occur in close conjunction with conflicts between the overlapping databases.

In 2004 our datasets were used in the First German Information Quality Contest [MMN05]. In the contest four interdisciplinary teams of Information-Quality-professionals had to identify systematic patterns in the contradicting data. The results show that there are no 'of the shelf products' to solve this task. Three of four teams applied association rule mining techniques (among others) to solve the problem. However, the shear amount of rules found by these techniques did not allow any further rule interpretation due to the limited time of two days for the contestants to present their results. Therefore we are convinced that a specialized method for finding contradiction patterns is justified. On the other hand, the usage of existing data cleansing tools by the competitors showed that there are several deficiencies in the data that cannot be revealed by our current mining method. For example, many conflicts in attribute *AUTHORS* are simply due to different ordering of author lists in the two databases. For the remaining conflicts many conflicting values show a very short edit distance (only 1 or 2 characters). Our current approach does not have the ability to consider such properties of conflicting values. Extending the mining algorithm to include specific properties of conflicting (and maybe even non-conflicting values) has to be considered as one possible direction of future work.

Chapter 5

Classification of Contradiction Patterns

Contradiction patterns highlight data characteristics together with conflicts between overlapping databases. Each pattern is regarded as an independent source of information about possible systematic conflict reasons. Pattern mining is global in that all the available information, i.e., attributes and conflict indicators, is taken into consideration. However, the presented mining algorithm does not allow any further constraints on conflict properties or attribute values. In particular, we only differ between the fact that there exists a conflict or not (indicated by the conflict indicator). This restriction is cumbersome in situations where we want to focus on certain classes of conflicts instead of treating all conflicts alike. Focusing on certain conflict classes is desirable since the different classes often require specific conflict resolution functions. We consider a set of conflicts sharing a specific property as a conflict class. Examples were given in the previous chapter e.g., different granularities for describing object properties or the usage of different vocabularies. In the first case, different values in one database appear in conflict with the same single value in the other database, i.e., there is a *1:N-mapping* between the conflicting values. In the second case, the mapping between the conflicting values is a *1:1-mapping*. Focusing on conflict classes yields additional information that is helpful for the expert user who is evaluating contradiction patterns for defining a conflict resolution strategy. In [MLF06a], we propose an approach to mining contradictory data that allows specification of certain constraints on conflicting values. The approach is based on the assumption that the contradicting databases are modified copies of a common ancestor database. Modification is represented by sequences of modification operations. These operations are the source of conflicts between the databases. In the following, we further motivate the suitability of such an ancestor-based conflict model. We present an algorithm for mining patterns that represent conflict generating modification operations from hypothetical modification sequences for a given pair of databases.

5.1 Reproducing Conflict Generation

Explanations for systematic conflicts given in the previous examples mainly consist of two parts: (i) a description of the condition under which the conflicts occur, and (ii) a description of the properties that the conflicting values obey. Consider the databases r_1 and r_2 in Figure 5-1 showing fictitious results of different research groups investigating a common set of individual owls. Database r_2 uses a finer grained coloring scheme for snowy owls than database r_1. Thus, conflicts occur under the condition

that the represented species is a snowy owl. The conflicting values themselves represent a *1:N-mapping* between the value 'white' in r_1 and the values 'white & grey' and 'snow-white' in r_2 suggesting different preciseness in the color naming schemes. Viewing databases r_1 and r_2 as modified copies of database r the conflicts in attribute *COLOR* are introduced by modifying r in that the color value is set to 'white' for all tuples having value 'Snowy Owl' in attribute Species for database r_1. This modification can be represented by a SQL-like update statement, e.g., `UPDATE r SET Color = 'white' WHERE Species = 'Snowy Owl'`. We call a model that considers a given pair of contradicting databases as modified copies of a common ancestor the *ancestor-based conflict model*. In the ancestor-based conflict model different sequences of SQL-like modification operations are used to introduce conflicts between a pair of databases. Figure 5-2 shows this scenario in more details, where databases r_1 and r_2 are modified descendants of a common, but unknown ancestor database. We call each sequence of operations that derives a database from an ancestor database the *modification sequence* for that particular database. Each operation in a modification sequence represents systematic data modifications and generates a new intermediate state onto the final (and given) databases. Figure 5-2 shows one of the intermediate states (r_p) for database r_1 in detail. Systematic conflicts between given databases in the ancestor-based conflict model are be represented by *condition-action pairs*. Each condition-action pair represents a potential operation in the modification sequence of one of the given databases.

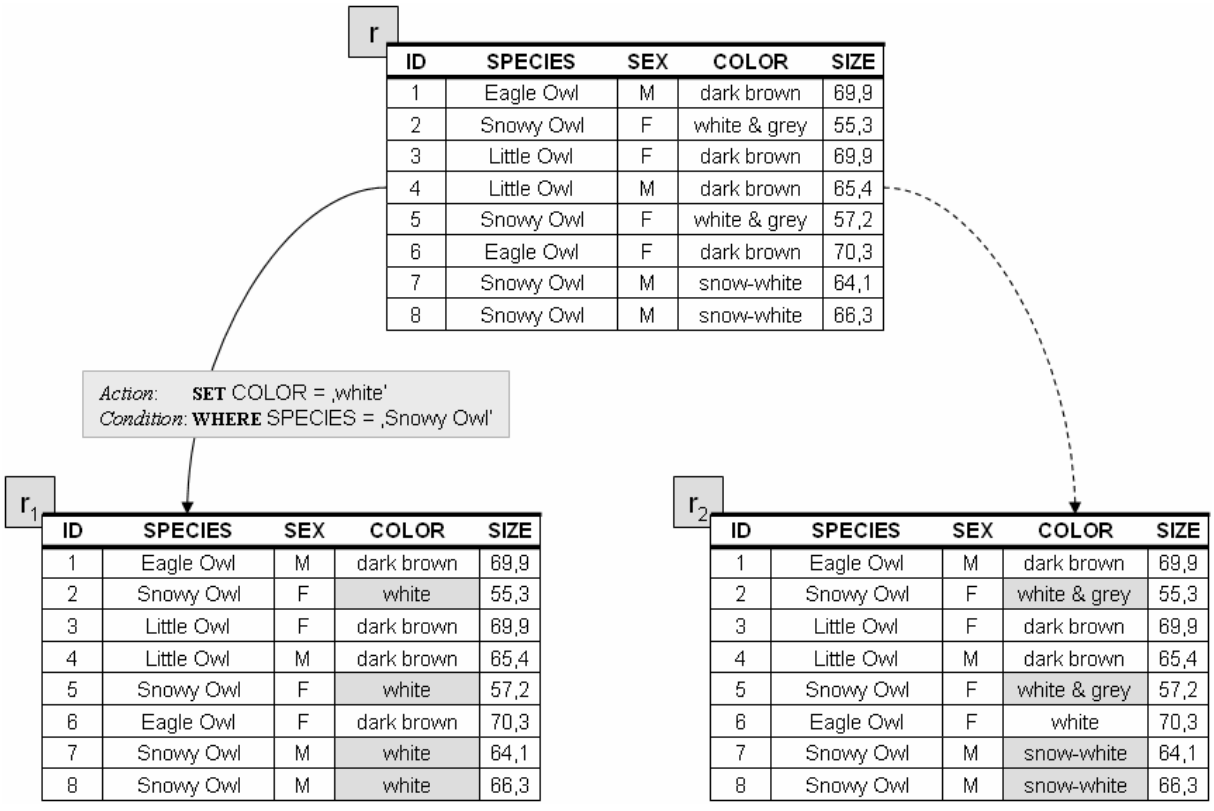
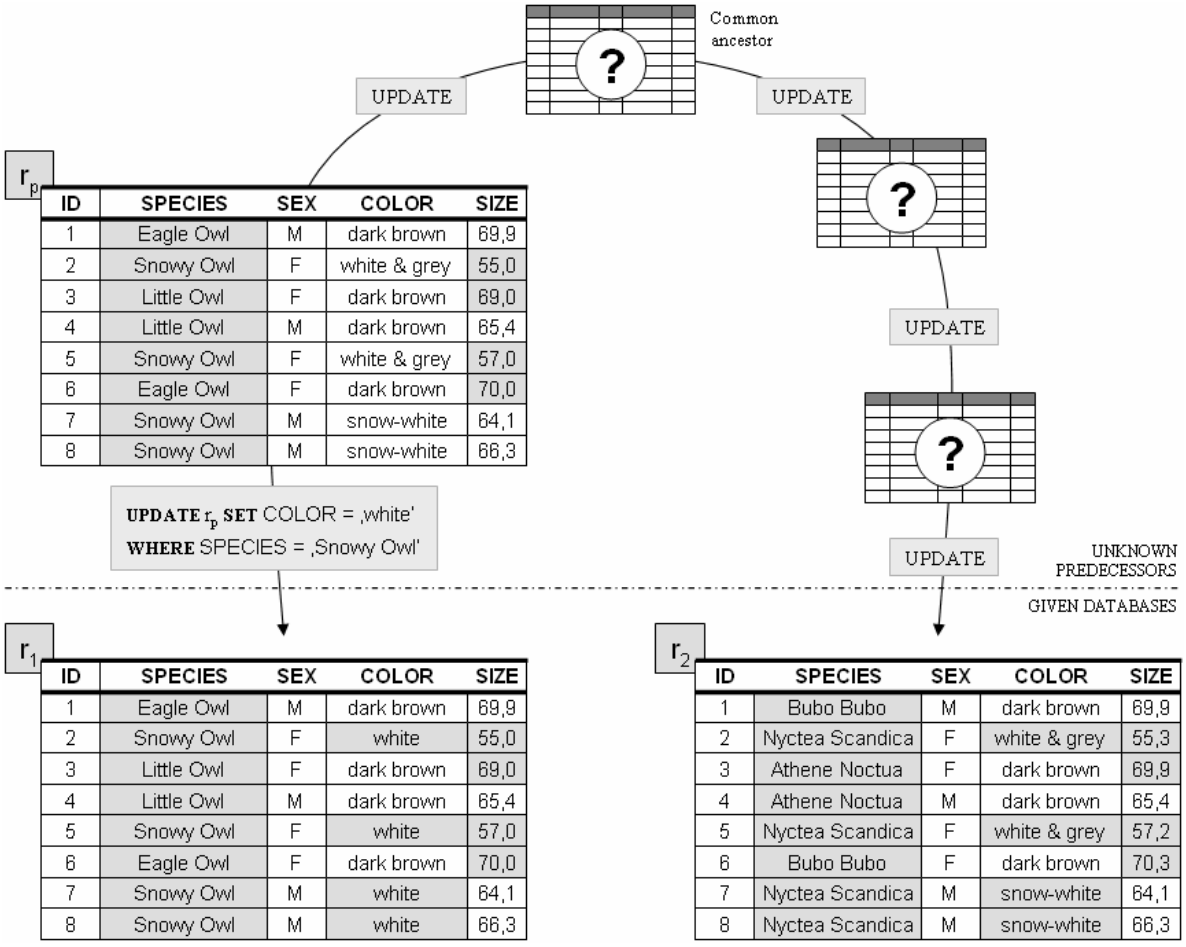


Figure 5-1: Contradicting databases r_1 and r_2 show fictitious results of different research groups investigating a common set of individual owls. Assuming that r_1 is a modified copy of r (while r_2 equals the original database), the conflicts in attribute *COLOR* are describable by a condition-action pair stating that the color is set to white for all snowy owls in r_1.

Systematic conflicts are well described by condition-action pairs and the ancestor-based conflict model is well suited for the identification of such descriptions. For instance, the model is a natural representation for the problem faced when integrating different protein structure data sources in COLUMBA. In COLUMBA we are faced with the problem of resolving conflicts between three overlapping data sources that are modified copies of the PDB flat-file data. Databases OPENMMS and E-MSD result from different cleansing efforts on the original data. Representing data cleansing workflows as a sequence of cleansing operations, we face a situation where different sequences of cleansing operations performed by different research groups independently modify the original PDB data. For conflict resolution we are interested in identifying unknown cleansing operations performed in each of the sequences. In a data cleansing scenario the common ancestor represents a database of low quality; the given descendants represent databases of improved quality. Gaining insights on the operations performed in each of the cleansing workflows helps in assessing the quality of the contradicting values. In conflict resolution, we want to retain quality-enhancing effects of cleansing operations in the resulting merged database. The ancestor-based conflict model is also useful to find differences resulting from systematic errors or differences in data production. Other than in the data cleansing case, the common ancestor in this scenario is not a physically existing database. Instead, the ancestor is considered a virtual database of high-quality that can be thought of as the ideal state. The given databases are different approximations of this state containing data deficiencies introduced by the data production process. Consider the databases r_1 and r_2 in Figure 5-2 which provides another example for differing results in investigating a common set of owls. Assume that database r in Figure 5-1 represents the ideal experimental results, i.e., the 'unknown' common ancestor database for r_1 and r_2 in Figure 5-2. Conflicts between r_1 and r_2 in attribute *SPECIES* are caused by using different vocabularies, i.e., English and Latin, to denote species names. These conflicts are introduced by executing a modification operation on r that applies a translation function on each tuple of r that translates from English to Latin species names (denoted as *MO1* in the following). Conflicts in attribute *COLOR* are due to the above mentioned finer grained color description for snowy owls (*Nyctea Scandica*) in database r_2 and are introduced by applying the modification operation shown in Figure 5-2 (*MO2*). Finally, conflicts within attribute *SIZE* are caused by truncating values for female owls of different species in database r_1 (*MO3*). Database r_1 is thus the result of a sequence of two modification operations <*MO2*, *MO3*> (the order is actually irrelevant in this example) applied to a copy of r. Database r_2 is the result of a single modification operation *MO1* being applied to another copy of r. Each of these modification operations represents either a transformation of data into a different format (*MO1*) or data deficiencies that result from the data production process (*MO2*, *MO3*).

For a given pair of databases the common ancestor is usually unknown (or non-existent) or unavailable to an expert user responsible for conflict resolution. The goal in reproducing conflict generation is to identify modification operations for a given pair of databases and a hypothetical common ancestor. Assuming that conflicts between databases follow an ancestor-based conflict model, identification of operations from the modification sequences in retrospective helps in assessing the quality of the databases. Depending on the interpretation of the modifications, we want to make sure that the effect of these operations is either contained in the resulting merged database (in the cleansing case) or excluded from the final data (in the production error case). In the following we define the concept of a preceding database that represents a possible intermediate state in the modification sequence of a given database. The preceding database is used in conflict generator mining to identify possible modification operations that represent systematic differences between two databases.



ID	SPECIES	SEX	COLOR	SIZE
1	Eagle Owl	M	dark brown	69,9
2	Snowy Owl	F	white & grey	55,0
3	Little Owl	F	dark brown	69,0
4	Little Owl	M	dark brown	65,4
5	Snowy Owl	F	white & grey	57,0
6	Eagle Owl	F	dark brown	70,0
7	Snowy Owl	M	snow-white	64,1
8	Snowy Owl	M	snow-white	66,3

ID	SPECIES	SEX	COLOR	SIZE
1	Eagle Owl	M	dark brown	69,9
2	Snowy Owl	F	white	55,0
3	Little Owl	F	dark brown	69,0
4	Little Owl	M	dark brown	65,4
5	Snowy Owl	F	white	57,0
6	Eagle Owl	F	dark brown	70,0
7	Snowy Owl	M	white	64,1
8	Snowy Owl	M	white	66,3

ID	SPECIES	SEX	COLOR	SIZE
1	Bubo Bubo	M	dark brown	69,9
2	Nyctea Scandica	F	white & grey	55,3
3	Athene Noctua	F	dark brown	69,9
4	Athene Noctua	M	dark brown	65,4
5	Nyctea Scandica	F	white & grey	57,2
6	Bubo Bubo	F	dark brown	70,3
7	Nyctea Scandica	M	snow-white	64,1
8	Nyctea Scandica	M	snow-white	66,3

Figure 5-2: The ancestor-based conflict model views a given pair of databases as copies of a common ancestor that are modified using different sequences of modification operations.

Preceding Databases

Reproducing conflict generation requires the identification of possible predecessors of the given databases. We consider exactly one predecessor for each of the databases r_1 and r_2 and each non-key attribute. The predecessor is used to describe how conflicts are introduced within the attribute by identifying modification operations that modify the predecessor resulting in conflicts between r_1 and r_2. Figure 5-2 shows the predecessor r_p for database r_1 and attribute *COLOR*. Also shown is a modification operation that describes the generation of conflicts by replacing the finer grained original color specifications in r_p with the more generic term 'white' in r_1. Currently, we only consider modification operations that modify the values within a single attribute.

Given a pair of databases r_1 and r_2, the set of potential predecessors is infinite. We restrict this set by allowing the values in the common ancestor to be modified at most once by any modification operation. This restriction enables us to define exactly one predecessor for each of the databases and each non-key attribute. In the remainder of this chapter we consider only conflicts within a fixed attribute $B \in R\backslash\{ID\}$ and a particular database r_1. Let r_p be the predecessor for database r_1 that is used to describe conflicts in attribute B. Database r_p equals r_1 in all attributes that are different from B. The values for r_p in attribute B are equal to the corresponding values in the contradicting database r_2. Database r_p is computed by a join of r_1 and r_2 on the primary key attribute and projection on the desired attributes, i.e., $r_p = \pi_{ATTLIST(B)}(r_1 \bowtie_{ID} r_2)$ with $ATTLIST(B) = \{r_1.A \mid A \in R \wedge A \neq B\} \cup \{r_2.B\}$.

We solely consider modification operations that alter values in a single attribute. Therefore, the values in attributes other that B are not affected by any operation that modifies attribute B. These values remain equal in r_p and r_1. The restriction that each value in the common ancestor is to be modified at most once implies that for corresponding values from r_1 and r_2 at least one of them is an original value. Obviously, values that are not in conflict have to be original values. For those values that are in conflict we assume the values in r_2 to be the original values. Since we are looking for modification operations that change values from r_p to r_1 the original values have to be those in r_2. For each tuple in r_p we further define a modifier.

Definition 5.1: For each tuple $t \in r_p$ we define a *modifier*, denoted by *modify*(t), as the value of attribute B for the according tuple $t_1 \in r_1$, i.e., $modify(t) = t_1[B]$, for $t \in r_p \wedge t_1 \in r_1 \wedge t_1[ID] = t[ID]$. ■

The modifier is the value that a tuple is modified with by a modification operation that describes the transformation of r_p into r_1. For values that are unchanged, values $t[B]$ and *modify*(t) are equal. For example, the modifier of tuple $t\{2\}$ in r_p from Figure 5-2 is 'white' and that of $t\{3\}$ is 'dark brown'. We use the modifier to describe the effect of modification operations and to classify conflict generators in the following.

Conflict Generators

We now define modification operations that describe the transformation of database r_p into r_1. We call these operations conflict generators as they introduce conflicts between the databases. Conflict generators are defined for a given database r_1, an attribute B, and the according predecessor r_p. Conflict generators aim at pointing towards possible modification operations in the generating modification sequences for database r_1.

Definition 5.2: A *conflict generator* is a (condition, action)-pair, denoted by (ρ, μ), where the pattern ρ acts as a condition that specifies a set of tuples in a preceding database that are modified and action μ describes the actual modifications. ■

Conditions in conflict generators are represented by special contradiction patterns as defined in the previous chapter. Conditions are contradiction patterns for attribute B and databases r_p and r_1. The important restriction for conflict generators is that these patterns are only allowed to contain terms from database r_p. Let *terms*(t) denote the set of terms for a tuple t. For each attribute $A \in R$ there exists a term (A, $t[A]$) $\in$ *terms*(t). A valid pattern for a conflict generator is then given by $\rho \subseteq \bigcup_{t \in r_p} terms(t)$.

Definitions of interestingness measures conflict potential, conflict relevance, and relevance deviation are not affected by this restriction. There is not always a single conflict generator describing all the conflicts in attribute B. Instead, we search for a set of conflict generators that represent possible systematic modifications. Using contradiction patterns as conditions in conflict generators enables us to ensure that the condition is characteristic for the occurrence of conflicts.

Actions in conflict generators are represented by relations that contain pairs of values from r_p and r_1 that are replaced with each other by the modification operation. Based on the tuples from r_p that satisfy condition ρ we define a relation $\mu(\rho(r_p)) \subseteq dom(B) \times dom(B)$ that reflects the modifications taken in conflict generation, i.e., $\mu(\rho(r_p)) = \{(t[B], modify(t)) \mid t \in \rho(r_p)\}$.

The relation μ contains exactly one element for every pair of corresponding values from r_p and r_1 for tuples in the given subset of r_p. The relation represents the mapping of values when transforming r_p

into r_1 and thereby the action performed by a modification operation. For example, the action taken by the update operation in Figure 5-2 defines a relation {('white & grey', 'white'), ('snow-white', 'white')}. We now classify conflict generators based on the properties of the relation they define. Based on this classification, we can focus on certain classes of conflicts in conflict generator mining.

5.2 Classification of Conflict Generators

Conflict generators represent a condition-action-based approach to conflict description and a different way of highlighting systematic conflicts between overlapping databases. Other than contradiction patterns, conflict generators focus on a single database and on conflicts in a single attribute only. Therefore, conflict generators do not allow any statements about the co-occurrence of conflicts in different attributes or characteristic properties over both databases. However, conflict generators explicitly focus on the conflicts in the considered attribute. This focus allows us to pose constraints on the conflicting values. Having knowledge about conflict generators in combination with knowing the values that are replaced by the action (i.e., the corresponding values in r_2), an expert user can infer easily for example that a pair of databases uses different naming schemes for certain object properties. A mining approach limited to one database is also helpful for defining 'a direction' of conflict generation, since conflict generators in many cases do not have an inverse function. The cause for the differences can therefore be linked to one of the particular sources. Based on the properties of the relation μ defined by the values of tuples satisfying ρ, we define a hierarchical classification for conflict generators as shown in Figure 5-3.

Systematic Conflict Generator: We call a conflict generator *systematic*, if the contradiction pattern ρ satisfies given conflict potential, conflict relevance, and relevance deviation thresholds. As discussed in the previous chapter, a contradiction pattern fulfilling these properties is assumed to be an adequate representation of systematic differences. In accordance, the conflict generator is assumed to represent systematic conflict generation in the ancestor-based conflict model.

Functional Conflict Generator: We call a systematic conflict generator *functional*, if the relation μ defines a function where each x relates exactly to one y. Regarding the description of conflicts, the action of a functional conflict generator is represented by a function f: $dom(B) \rightarrow dom(B)$. As shown in Figure 5-3, there is an adequate SQL-statement representing a functional conflict generator, i.e. `UPDATE` r_ρ `SET B = f(B) WHERE` ρ.

Injective Conflict Generator: A functional conflict generator is called *injective*, if different x values are always mapped to different y value, i.e., the function defines a *1:1-mapping* between the values. The action of an injective conflict generator is described by an injective function g: $dom(B) \rightarrow dom(B)$. An injective conflict generator can for example represent the translation of values between different vocabularies.

Non-Injective Conflict Generator: In cases where a functional conflict generator is not injective the function h: $dom(B) \rightarrow dom(B)$ defines an *N:1-mapping* between the values in r_p and r_1. Consequently, we call these functional conflict generators *non-injective*. It is obvious that a functional conflict generator is either injective or non-injective.

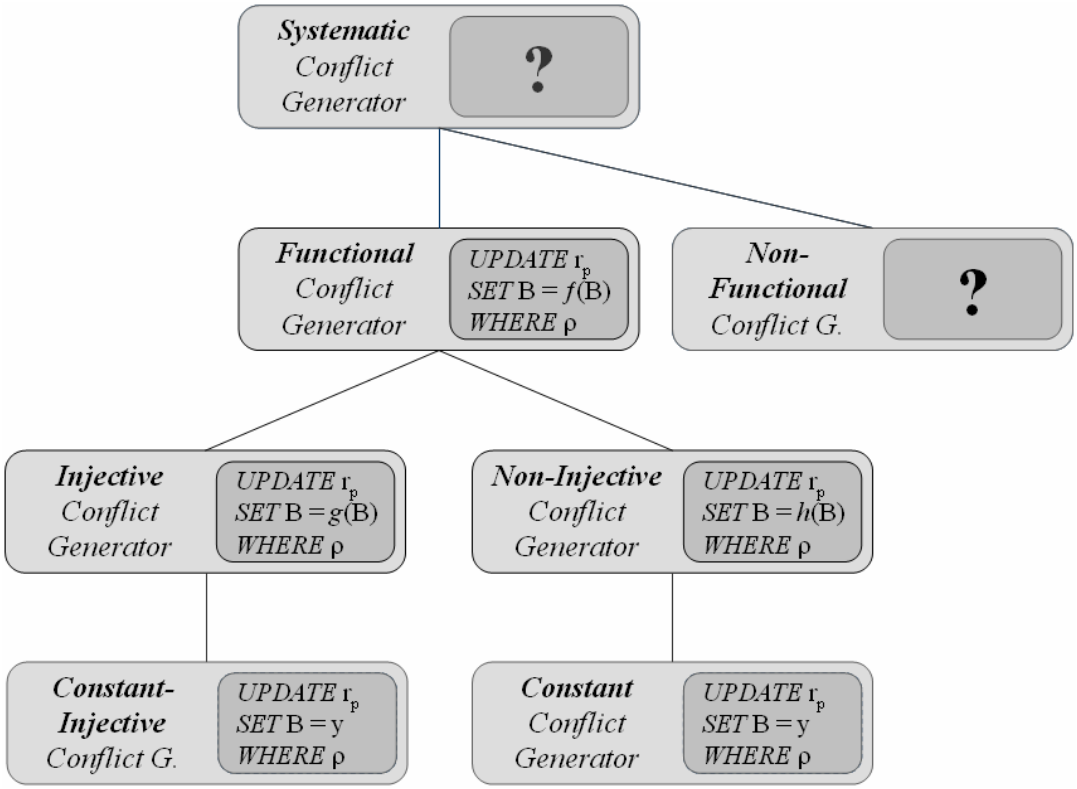


Figure 5-3: Classification hierarchy of conflict generators based on the relation of values they define. For most of the classes there is a simple SQL-like representation of the corresponding conflict generator.

Constant Conflict Generator: *Constant conflict generators* are a special cases of non-injective conflict generators in which the values from r_p are mapped onto a single constant value y. The conflict generator in Figure 5-2 is an example constant conflict generator that maps different values to the value 'white'.

Constant-Injective Conflict Generator: In cases where the relation μ contains only a single element we call the conflict generator *constant-injective*. These conflict generators represent for example the replacement of a single value in r_p with a single value in r_1.

Non-functional Conflict Generator: A systematic conflict generator is classified as being *non-functional*, if we cannot define a function that describes a valid mapping for relation μ. Although these conflict generators are systematic, they do not carry any additional information and are therefore regarded as simple contradiction patterns defined on database r_p. In case of a non-functional conflict generator, we are also unable to give a corresponding SQL-like statement for the conflict generator.

The described classification of conflict generators allows us to focus on certain classes during conflict generator mining. Our hierarchical classification of conflict generators defines seven different classes. For mining conflict generators we are interested in functional conflict generators only, since non-functional conflict generators are special cases of contradiction patterns. In the following algorithm we allow to constrain the returned conflict generators to the following classes, denoted by *F* for *functional* (and non-injective), *I* for *injective*, *C* for *constant*, and *I&C* for *injective and constant*, with $F \supseteq I$, $F \supseteq C$, and $I\&C = I \cap C$.

5.3 Mining Functional Conflict Generators

Mining conflict generators is accomplishable using contradiction pattern mining algorithms presented in Section 4.3.1. After mining contradiction patterns (restricted to the set of terms in database r_p), we prune all those patterns from the result that do not define a valid mapping for the desired conflict generator class. A drawback of this approach is that it does not allow pruning of contradiction patterns that do not define a valid mapping in advance. The reason is that our algorithm for contradiction pattern mining narrows the set of tuples that satisfy a candidate pattern by adding additional terms to the pattern. Thus, a candidate pattern violating a mapping constraint may satisfy the constraint after adding additional terms. In [MLF06a], we present an algorithm that mines conflict generators and allows pruning based on the mapping relation defined by the candidate patterns. The algorithm is based on a tuple enumeration approach as described in [CTX+04, PCT+03].

Mining conflict generators using tuple enumeration is based on the following property: Each set of tuples $s \subseteq r$ defines a pattern, denoted by *pattern*(s), that is the set of terms common to all tuples in s, i.e., $pattern(s) = \bigcap_{t \in s} terms(t)$. If *pattern*($s$) is not empty it represents a closed pattern. For example, tuples $t\{2\}$ and $t\{8\}$ in database r_p in Figure 5-2 define the closed pattern {(*SPECIES*, 'Snowy Owl')}. This pattern is also satisfied by tuples $t\{5\}$ and $t\{7\}$. However, we cannot add another term to the pattern without changing its support in r_p. For example, if we add another term to {(*SPECIES*, 'Snowy Owl')} either tuple $t\{2\}$, tuple $t\{8\}$, or both would not satisfy the new pattern. As a consequence, each set of tuples that share at least one common term defines a closed pattern and different tuple sets may define the same closed pattern.

An algorithm that extends sets of tuples allows to test immediately whether an additional tuple violates the current mapping defined by the tuples in a given set or not. Therefore, tuple enumeration allows pruning conflict generators based on their value mapping. Algorithms like CARPENTER [PCT+03] and FARMER [CTX+04] efficiently enumerate those sets of tuples that define the complete set of closed patterns that satisfy a given support threshold. Our algorithm for Retrospective Conflict Generator Mining (RECOGNIZE) adopts the CARPENTER algorithm and extends the pruning capabilities to avoid enumeration of tuple sets that do not represent conflict generators of the requested class. RECOGNIZE takes as parameters database r_p, conflict potential and conflict relevance thresholds, and the requested class for the returned conflict generators (*class*), i.e., *F*, *I*, *C*, or *I&C*. Figure 5-4 shows the main algorithm. In contrast to contradiction pattern mining based on term enumeration, REGONIZE is not able to prune contradiction patterns based on their relevance deviation threshold. The algorithm therefore does not consider a relevance deviation threshold. If we do require such a threshold for the patterns in conflict generators, we have to use the variation of the contradiction pattern mining algorithm as described above.

```
1 REGOGNIze (r_p, min_pot, min_rel, class) {
2   CG := {};
3   s_B := {t | t ∈ r_p ∧ t[B] ≠ modify(t)};
4   tupsup := min_rel * |s_B|;
5   minePattern({}, s_B, tupsup, r_p, CG, min_pot, class);
6   return CG;
7 }
```

Figure 5-4: The REGONIZE algorithm for mining conflict generators using tuple enumeration.

```
1 minePattern(s_b, s_e, tupsup, r_p, CG, min_pot, class) {
2   s_e':={t | t ∈ s_e ∧ t[ID] > max(s_b) ∧ terms(t) ∩ pattern(s_b) ≠ {}
      ∧ compatible(t, s_b)};
3   if (|s_e'| + |s_b| < tupsup) {
4     return;
5   }
6   s_y:={t | t ∈ s_e' ∧ terms(t) ⊇ pattern(s_b)};
7   if (¬validMapping(s_y, class)) {
8     return;
9   }
10  if (pattern(s_b) ∈ CG) {
11    return;
12  }
13  if ((|s_b| + |s_y| ≥ tupsup) ∧ (pot(pattern(s_b)) ≥ min_pot) ∧
      (validMapping(pattern(s_b)(r_p), class))) {
14    CG:=CG ∪ pattern(s_b);
15  }
16  for each t ∈ (s_e' - s_y) do {
17    minePattern(s_b ∪ {t}, (s_e' - s_y) - {t}, r_p, CG, min_pot, class);
18  }
19}
```

Figure 5-5: Tuple enumeration is performed recursively in subroutine *minePattern*.

Let s_B denote the subset of r_p containing those tuples that have a conflict in attribute B. The algorithm enumerates all subsets $s_b \subseteq s_B$ that (i) have sufficient size to satisfy the relevance threshold, i.e., $|s_b| > min_{rel} * |s_B|$, (ii) whose resulting pattern $pattern(s_b)$ satisfies the conflict potential threshold, and (iii) whose mapping $mapping(s_b)$ represents a valid mapping based on the specified mapping class. The enumeration is done using subroutine *minePattern* shown in Figure 5-5. The parameters for *minePattern* are the current tuple set s_b, the set of tuples s_e that are considered as possible extensions for s_b, database r_p, the result set CG, the conflict potential threshold, and the requested conflict generator class.

We assume that elements in tuple sets are sorted in ascending order of their primary key. Tuple sets are enumerated in depth first order based on the primary key to avoid the enumeration of duplicate tuple sets (Figure 5-6 shows an example depth first order enumeration). In the first step of subroutine *minePattern* the candidate tuples from s_e for extending s_b are determined (*lines 2-5*). These are the tuples that contain at least one of the terms in $pattern(s_b)$, i.e., that satisfy at least one subset of $pattern(s_b)$. To ensure depth first enumeration, the candidate tuples have to have a primary key that is greater than the maximum primary key (returned by function *max*) of tuples in s_b. For example, the tuple set $\{t_1, t_2\}$ in Figure 5-6 defines the pattern $\{\tau_1, \tau_2, \tau_3\}$ and can be extended with tuples t_3 and t_4 as both t_3 and t_4 for example contain term τ_2. In addition to the original CARPENTER algorithm we also request that the value pair $(t[B], modify(t))$ is compatible with the mapping defined by $pattern(s_b)$ regarding the requested class (details below). If the sum of tuples in s_e' and in s_b is below the minimal tuple support (*tupsup*) we return, because the tuple set s_b cannot be extended to form a relevant conflict generator.

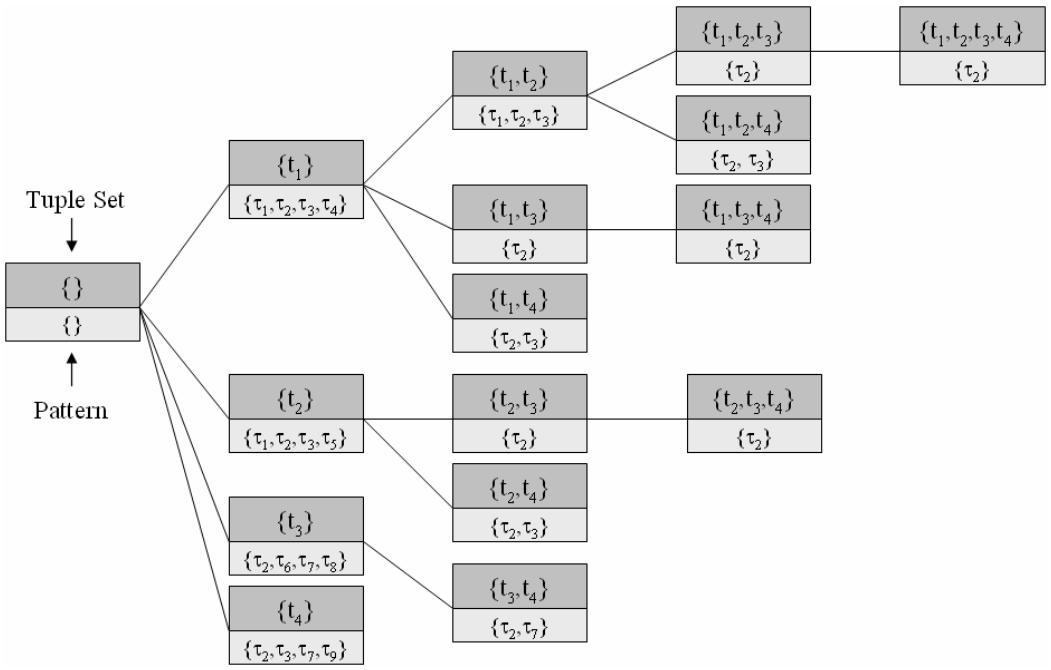


Figure 5-6: An example for enumerating tuple sets in a depth-first manner. Each node in the enumeration tree contains a tuple set and a pattern that is defined by the terms that are common to all tuples in the respective set.

The second step of subroutine *minePattern* determines the subset of candidate tuples s_y that satisfy *pattern*(s_b), i.e., contain all terms in the pattern (*lines 6-9*). It follows that $s_b \cup s_y$ defines the set of tuples from s_e that satisfy *pattern*(s_b). In REGONIZE there is an additional ability for pruning if the tuples in s_y do not define a mapping that is valid for the desired class. The tuples in s_y are later excluded as further extension candidates of s_b as these extensions would only generate identical closed patterns (*line 16*).

In the last step of subroutine *minePattern*, we first check whether *pattern*(s_b) is already contained in *CG* (*lines 10-12*). In case the pattern is not contained, we add *pattern*(s_b) to *CG* if all three constraints as listed above are satisfied, i.e., sufficient conflict relevance, sufficient conflict potential threshold, and defining a valid mapping based on the specified mapping class (*lines 13-15*). We then extend the current tuple set using the remaining candidates in s_e' - s_y and call *minePattern* recursively in order to build the complete tuple set enumeration tree.

The subroutine *validMapping* checks whether a set of tuples defines a valid mapping regarding *class*. This check is trivial for either of the classes *C* or *I&C* where we simply test for equality of the modification values (and the values $t[B]$ in case of *I&C*) for the elements in μ. The case of *class* being *F* or *I* is described in the following: For each element $(x, y) \in \mu$, referred to as mapping term in the following, we maintain a list of incompatible mapping terms, denoted by *incomp*((x, y)). A pair of mapping terms (x_1, y_1) and (x_2, y_2) is considered incompatible for conflict generators of class *F*, if $x_1 = x_2$ and $y_1 \neq y_2$. The mapping terms are incompatible for class *I*, if they are incompatible for class *F* or $x_1 \neq x_2$ and $y_1 = y_2$. For *validMapping*(*s*) to be true, we request that $\bigcap_{t \in s}$ *incomp*(($t[B]$, *modify*(t))) = {}, i.e., there exist not incompatibilities between the mapping terms of the tuples in a given tuple set s. In subroutine *compatible* we request *incomp*(($t[B]$, *modify*(t))) to be disjoint with $\bigcup_{t \in s}$ *incomp*(($t[B]$, *modify*(t))), i.e., the mapping term is not incompatible with any of the mapping terms currently in *s*.

In Figure 5-2 there are three functional conflict generators for a relevance threshold of 50%, representing conflicts for female snowy owls, male snowy owls, and snowy owls in general. They all have conflict potential of 100%. The first two conflict generators belong to class *I&C* and the other belongs to *C*. Experiments with data sets of protein structures as used in Section 4.4 show that an average of 63% of the patterns for each attribute represent functional conflict generators, with *C* and *I&C* being the most frequently subclasses.

5.4 Summary and Related Work

Within this chapter we present a modified approach towards highlighting systematic differences between overlapping databases. The approach is based on conflict generators that are (condition-action)-pairs and represent a retrospective way of describing differences based on the ancestor-based conflict model. The conditions describe characteristic data patterns that hold in conjunction with conflicts. Actions give a value mapping that summarizes the conflicting values. This bipartite way of describing systematic differences not only highlights data characteristics in conjunction with conflicts, but also gives information about conflicting values. Based on properties of the mappings, we are enabled to specify additional constraints on the patterns we are interested in. Functional conflict generators can be represented as SQL-like modification operations. These operations are the most common way of modifying relational databases. We present an algorithm that allows mining for conflict generators with restrictions to the class of the value mapping in their action part. We differ between four classes of mappings that represent certain conflict classes, e.g., translation of values or different specificity for describing object properties. The presented algorithm can be extended to allow further restrictions on the conflicting values. We consider mining conflict generators with additional constraints as future work.

Conflict generators, in contrast to contradiction patterns, are not global in that they do not take all the available information into consideration. Instead, conflict generators focus only on one of the databases and the conflicts in one particular attribute. When mining the complete set of conflict generators, we have to execute algorithm REGOGNIZE for each non-key attribute and each of the contradicting databases. The resulting descriptions of systematic differences are still independent of each other as it is the case with contradiction patterns. We give up on this independence in the last part of this thesis where we describe algorithms that define sequences of update operations describing the complete set of differences between a pair of contradicting databases.

The body of related work to conflict generator mining is limited. *Weiguo Fan et al*. [FLMC01] present a method for finding patterns in contradictory data to support conflict solution. Their focus is the identification of rules that describe the conversion of contradicting values. The authors do not request these rules to be associated with a descriptive condition as in our approach. On the other hand, we do not consider the identification of complex data conversion rules. However, the mappings defined by conflict generators could be used as input for the methods described in [FLMC01]. There is also a large body of work on statistical data editing, i.e., the automatic correction of conflicting values, based on the FELLEGI-HOLT-MODEL [FH76]. These approaches rely on edits (rules) for conflict detection and determine the minimal number of changes necessary for conflict elimination. In contrast, we use object identifiers for conflict identification and currently do not consider automatic value modification.

Part III

On the Distance of Databases

Chapter 6

Update Distance of Databases

Contradiction patterns and conflict generators provide valuable information for conflict resolution by highlighting systematic differences between contradicting databases. The algorithms presented in the second part of this thesis mine patterns for each attribute independently. An expert user evaluating the patterns has to identify those that are best applicable to define a conflict resolution strategy. In [MFL06, MLF06b], we develop a different approach for finding regularities in contradicting databases: The detection of minimal set-oriented update sequences. Just as conflict generators, minimal update sequences use SQL-like update operations to highlight systematic differences. These set-oriented operations are commonly used for modification of relational databases and are therefore well suited for conflict descriptions. Furthermore, update sequences highlight systematic differences for the whole database and not only for a particular attribute. Update sequences thereby enable identification of dependencies between conflicts in different attributes that are not revealed by contradiction patterns.

Our idea of using minimal update sequences is best explained by analogy with the usage of the string edit distance [Gus97] in biological sequence analysis (see Figure 6-1). The DNA sequence of a gene is a string over a four letter alphabet. To learn about the function of a specific gene in a specific species, biologists search for evolutionary related genes of known function in other species. This evolutionary relatedness (or distance) is proportional to the number of evolutionary events that have occurred to the sequence of a common ancestor, deriving the observed sequences, which in turn is proportional to the number of evolutionary events that would be necessary to turn one gene into another. Using a simple model of evolution encompassing only changes, deletions, and insertions of single bases (i.e., characters of the sequence), the number of evolutionary events is measured by the edit distance between two gene sequences, i.e., the minimal number of edit operations (or evolutionary events) that transform one string into the other. Similarly, we consider SQL-like insertions, deletions, and modifications of tuples as the fundamental operations for the manipulation of data stored in relational databases. Thus, to assess the “evolutionary relationship” of two databases, we propose to use the minimal number of such operations that turn one database into the other. We call this number the *update distance* between two databases. The number of possible sequences that transform one database into another is large (the most obvious sequence changes each conflict individually using a single update operation). Each sequence of operations as long as the update distance is one of the simplest possible explanations for the

observed differences. Following the "*Occam's Razor*" principle, we conclude that the simplest explanations are also the most likely.

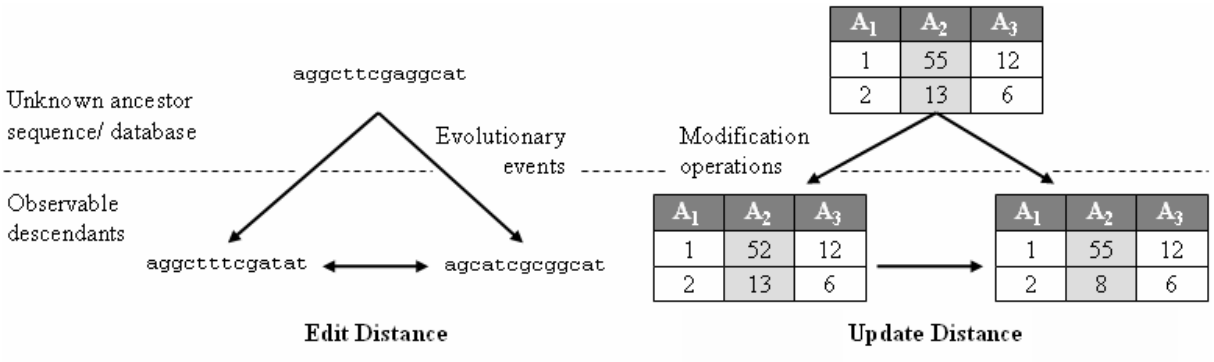


Figure 6-1: The analogy of using edit distance to discover evolutionary relationship between biological sequences (left) and update distance to discover regularities in contradicting databases (right).

The update distance is a semantic distance measure, as it is inherently process-oriented in contrast to purely syntactic measures such as counting differences. Minimal update sequences are in accordance with the ancestor-based conflict model. We will use the ancestor-based conflict model as foundation for the description of systematic conflicts in the remainder of this thesis. For a pair of databases r_1 and r_2 that are descendants of a common ancestor r_a transforming r_1 into r_2 requires to (a) redo modifications that occur in the modification sequence from r_a to r_2, and (b) undo modifications in the modification sequence from r_a to r_1. Therefore, minimal update sequences give valuable clues on what has happened to a pair of database to make them different from their original state. In this chapter, we present an exact algorithm for computing the update distance and for finding minimal sequences of update operations for a pair of databases. Even though we consider only a restricted form of updates (namely those where the attribute values are set to constants), our algorithms for computing the exact solution require exponential space and time.

Example 6-1: To give an idea of the complexity of the problem, consider databases r_1 and r_2 in Figure 6-2. Clearly, the update distance for r_1 and r_2 can be determined by enumerating update sequences of increasing length until one sequence is found that implements all necessary changes. This would generate 294,998 intermediate states. An intuitive idea to prune the search space would be to use a greedy strategy, i.e., to select at each stage the operation that solves the most conflicts. This strategy reduces the number of generated intermediate states to 42. The shortest sequence found using such an approach has four elements (update sequence a)), although the update distance between databases r_1 and r_2 is three (update sequence c)). Another pruning idea might be to avoid modification operations that introduce new conflicts. This results in only 32 generated intermediate states. However, using this heuristic worsens the result, as now the shortest sequence is of length five (update sequence b)). Intuitively, it is often necessary to use operations that in first place introduce new values (and thereby new conflicts) that can be used as discriminating conditions in later update operations. The first operation in update sequence c) temporarily increases the total number of conflicts, but this is compensated in later operations that are now able to solve more conflicts within one statement. ■

In the following, we build the necessary vocabulary for the definition of update distance and derive upper and lower bounds that are important for optimization. Based on these definitions, we describe

our algorithm for calculating minimal update sequences for a given pair of databases. Heuristics and problem variations to improve the efficiency of the algorithm are discussed in Chapter 7.

r_1

A_1	A_2	A_3
1	1	1
2	1	1
3	1	1
4	2	1
5	3	1
6	4	1
7	5	1
8	6	1
9	1	0
10	1	0

r_2

A_1	A_2	A_3
1	1	1
2	1	1
3	1	1
4	2	0
5	3	0
6	4	0
7	5	0
8	6	0
9	1	0
10	1	0

a)
```
UPDATE r1 SET A3=0
UPDATE r1 SET A3=1 WHERE A1=1
UPDATE r1 SET A3=1 WHERE A1=2
UPDATE r1 SET A3=1 WHERE A1=3
```
b)
```
UPDATE r1 SET A3=0 WHERE A1=4
UPDATE r1 SET A3=0 WHERE A1=5
UPDATE r1 SET A3=0 WHERE A1=6
UPDATE r1 SET A3=0 WHERE A1=7
UPDATE r1 SET A3=0 WHERE A1=8
```
c)
```
UPDATE r1 SET A3=2 WHERE A2=1
AND A3=1
UPDATE r1 SET A3=0 WHERE A3=1
UPDATE r1 SET A3=1 WHERE A3=2
```

Figure 6-2: An example for the need to introduce conflicts in order to find an optimal solution.

6.1 Distance Measures for Databases

Without loss of generality, we assume $dom(A) = \mathbb{N}$ for all attributes $A \in R$. In contrast to the previous chapters, we now consider unmatched tuples in overlapping databases, i.e., tuples in either r_1 or r_2 without a matching partner in the other database.

Definition 6.1: Given a pair of databases r_1 and r_2. The *set of unmatched tuples* from r_1, denoted by $U(r_1, r_2)$, is defined as the set of tuples that do not have a matching partner in r_2, i.e., $U(r_1, r_2) = r_1 / (r_1 \ltimes_{ID} r_2)$. ■

We use the semi-join $r_1 \ltimes_{ID} r_2$ on the primary key to determine the matching tuples in r_1 and r_2.

Conflicts between overlapping databases are represented by the matching pair m and the attribute A in which the conflict occurs.

Definition 6.2: The *set of conflicts* between a pair of databases r_1 and r_2, denoted by $C(r_1, r_2)$, is the set of all tuples (m, A) where a conflict in attribute A of pair m exists, i.e., $C(r_1, r_2) = \{(m, A) \mid (m, A) \in M(r_1, r_2) \times R \wedge conflict(m, A)\}$. ■

6.1.1 Transformers for Pairs of Databases

Update operations are used to modify existing databases. For relational databases there are three types of basic update operations, namely insert, delete, and modify [Vos91]. An insert operation creates a new tuple. A delete operation removes a set of tuples satisfying a given selection criteria. A modification operation changes the value for an attribute within a set of tuples satisfying a given selection criteria. We use patterns and terms as defined in Definition 4.5 and Definition 4.6 in our definition of update operations. In general, update operations can be considered as functions that map databases onto each other. Let $\mathfrak{R}(R)$ denote the infinite set of databases following schema R that satisfy the primary key constraint.

Definition 6.3: An *update operation* ψ over schema R is a mapping $\psi : \mathfrak{R}(R) \rightarrow \mathfrak{R}(R)$. We distinguish between three types of update operations:

- The *insert operation*, denoted by ψ_ι, is a n-tuple $(\tau_1, \ldots, \tau_n)$. It contains exactly one term for each of the attributes A_i from R. It adds a new tuple t_{new} to r, with $t_{new}[A_i] = value(\tau_i)$ for $1 \leq i \leq n$. If there already exists a tuple t in r with $t[ID] = t_{new}[ID]$, the database remains unchanged. Otherwise, the result of $\psi_\iota(r)$ is $r \cup \{t_{new}\}$.
- The *delete operation*, denoted by ψ_δ, is defined by a single pattern ρ. It removes all tuples from a relation, that satisfy the pattern ρ, i.e., $\psi_\delta(r) = r \,/\, \rho(r)$.
- The *modification operation*, denoted by ψ_μ, is a term-pattern pair (τ, ρ). We exclude key attributes from being modified. Therefore, $attr(\tau)$ is element of $R \,/\, ID$. A modification operation modifies all tuples within a relation that satisfy ρ. For these tuples, the value for attribute $attr(\tau)$ is set to $value(\tau)$. ■

Given a modification operation $\psi_\mu = (\tau, \rho)$, we refer to τ as the *modification term*, to $value(\tau)$ as the *modification value*, to $attr(\tau)$ as the *modified attribute*, and to ρ as the *modification pattern*. Note that there not necessarily exists a reverse operation for each modification operation. For example, the operation $\psi_\mu = ((A_2, 7), \{(A_3, 1)\})$ sets the value for attribute A_2 to 7 for the tuples $t\{1\}, \ldots, t\{8\}$ when applied to database r_1 in Figure 6-2. We need at least six modification operations to undo this single operation. There is also no single reverse operation for delete operations that delete more than one tuple. We now have all the tools at hand to define minimal sequences of update operations.

Definition 6.4: An *update sequence* $\Psi = <\psi_1, \ldots, \psi_k>$ is an ordered list of update operations. Applied on a database r_1, an update sequence *generates* (or *derives*) a database $r_2 = \Psi(r_1)$ by executing the update operations in given order on r_1, i.e., $\Psi(r_1) = \psi_k(\ldots(\psi_1(r_1))\ldots)$. ■

The databases that are generated by the update operations of an update sequence while transforming r_1 into r_2 are called *intermediate states*. Obviously, the order of operations within an update sequence is important. For example, the update sequences $\Psi_1 = <\psi_{\mu 1}, \psi_{\mu 2}>$ and $\Psi_2 = <\psi_{\mu 2}, \psi_{\mu 1}>$ with $\psi_{\mu 1} = ((A_2, 7), \{(A_3, 1)\})$ and $\psi_{\mu 2} = ((A_3, 7), \{(A_3, 1)\})$ have different results when applied to database r_1 in Figure 6-2. The first sequence results in a database where the value for attribute A_2 and A_3 is 7 in tuples $t\{1\}$,

…, $t\{8\}$. In the second sequence the operation $\psi_{\mu 1}$ has no effect, as its pattern is no longer satisfied by any of the tuples after applying operation $\psi_{\mu 2}$.

Definition 6.5: We call Ψ a *transformer* for databases r_1 and r_2, iff $\Psi(r_1) = r_2$. ■

The number of update operations within a sequence is called its length and is denoted by $|\Psi|$. Figure 6-2 lists three update sequences of different length that are transformers for the databases r_1 and r_2.

Definition 6.6: An update sequence Ψ is called a *minimal transformer* for a pair of databases r_1 and r_2, if $\Psi(r_1) = r_2$ and there does not exist another transformer Ψ' with $\Psi'(r_1) = r_2$ and $|\Psi'| < |\Psi|$. ■

There may be several minimal transformers for a pair of databases r_1 and r_2. The set of all minimal transformers for r_1 and r_2 is denoted as $T(r_1, r_2)$.

Example 6-2: For the pair of databases r_1 and r_2 with $\Delta_U(r_1, r_2) = 2$ and $\Delta_U(r_2, r_1) = 3$ four minimal transformers (in SQL-like notation) are shown, two from $T(r_1, r_2)$ and two from $T(r_2, r_1)$ respectively.

r_1

A_1	A_2	A_3	A_4
1	2	1	1
2	1	2	1
3	1	2	0
4	1	2	1

r_2

A_1	A_2	A_3	A_4
1	2	1	3
2	2	2	3
3	2	2	3
4	2	2	3

$T(r_1, r_2)$ examples:

```
Ψ1)  UPDATE SET A4 = 3
     UPDATE SET A2 = 2 WHERE A2 = 1
Ψ2)  UPDATE SET A2 = 2
     UPDATE SET A4 = 3
```

$T(r_2, r_1)$ examples:

```
Ψ1)  UPDATE SET A2 = 1 WHERE A3 = 2
     UPDATE SET A4 = 1
     UPDATE SET A4 = 0 WHERE A1 = 3
Ψ2)  UPDATE SET A4 = 0 WHERE A1 = 3
     UPDATE SET A4 = 1 WHERE A4 = 3
     UPDATE SET A2 = 1 WHERE A3 = 2
```

■

We now define distance measures for databases to quantify their similarity. Such a measure is represented by a distance function, which assigns a non-negative value to a pair of databases, with a smaller value, i.e., a shorter distance, reflecting a greater similarity. Similar to existing distance measures for strings, which rely on the edit operations insert, delete, and replace, we use sequences of update operations in our definitions.

6.1.2 The Resolution Distance

An obvious distance measure for a pair of databases is the total number of differences between them. This number is given by the sum of unmatched tuples and conflicts between these databases. Such a distance measure reflects the maximal number of necessary update operations for transforming the databases into each other.

Definition 6.7: For a pair of databases r_1 and r_2, the *resolution distance* $\Delta_R(r_1, r_2)$ is defined as the sum of the number of unmatched tuples in either database and the number of conflicts between the databases, i.e., $\Delta_R(r_1, r_2) = |U(r_1, r_2)| + |U(r_2, r_1)| + |C(r_1, r_2)|$. ■

For the databases r_1 and r_2 in Example 6-2, the resolution distance is 7, which is equal to the number of conflicts between the databases r_1 and r_2. It follows, that $\Delta_R(r_1, r_2) = \Delta_R(r_2, r_1)$, as $C(r_1, r_2)$ equals $C(r_2, r_1)$. The resolution distance, however, is not a metric, as the triangle inequality $\Delta_R(r_1, r_2) + \Delta_R(r_2, r_3) \geq \Delta_R(r_1, r_3)$ does not hold. For example, a tuple occurring within r_1 and r_3 but not in r_2 counts only twice on the left side of the inequality but potentially $(|R| - 1)$-times on the right side, because there may occur a conflict within every non-key attribute of the corresponding matching pair.

Lemma 6.1: For each pair of databases r_1 and r_2 there exists a transformer Ψ of length $\Delta_R(r_1, r_2)$.

Proof: In order to transform r_1 into r_2, we have to (i) remove the tuples from r_1 without a matching partner in r_2, (ii) solve the conflicts within the matching pairs, and (iii) insert those tuples that exist in r_2 but not r_1. Due to the primary key property every tuple t from database r is individually selectable by a pattern $\rho = \{(ID, t[ID])\}$. As the primary key is unchangeable, this is always true for any existing tuple. The deletions are accomplished using a single delete operation for every unmatched tuple in r_1, i.e., for every tuple in $U(r_1, r_2)$. The conflicts are solved using a single modification operation for every element $(A, m) \in C(r_1, r_2)$, with the modification term $\tau = (A, tup_2(m)[A])$ and the pattern $\rho = \{(ID, id(m))\}$. The inserts are performed by executing an insert operation on r_1 for every unmatched tuple from r_2, i.e., for every tuple in $U(r_2, r_1)$. Overall, this requires $|U(r_1, r_2)|$ delete operations, $|C(r_1, r_2)|$ modification operations, and $|U(r_2, r_1)|$ insert operations. Any sequence of these operations is a transformer for r_1 and r_2. ■

6.1.3 The Update Distance

The described transformers do not necessarily reflect the optimal solution regarding the number of update operations needed to transform one database into another. Often, there is the possibility to solve more than one conflict using a single modification operation. The same is true for multiple deletes in order to minimize the overall number of necessary operations. The possibility of solving multiple conflicts at once is reflected in the definition of the following distance measure considering update operations that affect an arbitrary number of tuples.

Definition 6.8: For a pair of databases r_1 and r_2, the *update distance* $\Delta_U(r_1, r_2)$ is defined as the length of any minimal transformer for r_1 and r_2. ■

Note that the update distance is also not a metric as it is not a symmetric relation, i.e., $\Delta_U(r_1, r_2)$ is not necessarily equal $\Delta_U(r_2, r_1)$. We consider the minimal transformers as explanations for observed differences between two databases. In order to avoid meaningless (or trivial) update sequences like (1) *delete all tuples in* r_1, and then (2) *for each tuple in* r_2 *perform an insert operation*, we further restrict the valid update operations within the transformers.

Definition 6.9: For any intermediate state r_i in the process of transforming r_1 into r_2 an operation ψ is *valid*, if $\psi(r_i) \neq r_i$ and:

- ψ is an insert operation, with $t_{new} \in r_2 / (r_2 \ltimes_{ID} r_1)$,

- ψ is a delete operation, where $\rho_\delta(r_i) \subseteq r_1 / (r_1 \ltimes_{ID} r_2)$, or
- ψ is a modification operation. ■

According to Definition 6.9, we allow inserts only for tuples from r_2 that do not have a matching partner in r_1 and deletions for tuples in r_1 that haven't got a matching partner in r_2. Modification operations are unrestricted.

The resolution distance and the update distance both describe sequences of update operations for transforming one database into the other. They differ, however, in the set of utilized update operations. Unfortunately, there exists no easy formula for calculating the update distance as there exists one for the resolution distance. An algorithm to determine the update distance and the set of all minimal transformers for a given pair of databases is described in Section 6.2. We prove in Section 7.2 that the problem of computing the set of minimal transformers is NP-hard when using only a restricted set of operations. While the calculation of the update distance is non-trivial, we can define upper and lower bounds.

Lemma 6.2: An upper bound for the update distance between a pair of databases r_1 and r_2, denoted by $UB(r_1, r_2)$, is given by the resolution distance $\Delta_R(r_1, r_2)$.

Proof: Due to Lemma 6.1, there exists a transformer of length $\Delta_R(r_1, r_2)$ for r_1 and r_2. Any transformer of length greater than the resolution distance is therefore not minimal. ■

To define a lower bound, we make use of our definition that each modification operation modifies only one attribute. We subsume the conflicts that are potentially solvable using a single modification operation within a conflict group.

Definition 6.10: Given a pair of databases r_1 and r_2 and a matching pair $m \in M(r_1, r_2)$. The *solution* of an existing conflict $(m, A) \in C(r_1, r_2)$ is given by the value $tup_2(m)[A]$ that has to be used as modification value in a modification operation to solve the conflict when transforming r_1 into r_2. ■

Definition 6.11: A *conflict group* κ is an attribute-value pair (A, x) with $attr(\kappa) = A \in R$ and $value(\kappa) = x \in dom(A)$. A conflict group represents the subset of conflicts $(m, A) \in C(r_1, r_2)$ having property $\kappa(r_1, r_2) = \{(m, A) \mid (m, A) \in C(r_1, r_2) \wedge attr(\kappa) = A \wedge value(\kappa) = tup_2(m)[A]\}$. ■

All conflicts represented by a conflict group κ occur in the same attribute A and have the same solution x. Hence, these conflicts are solvable using a modification operation with κ as the modification term. Let $K(r_1, r_2)$ be the set of all conflict groups between a pair of databases. There are two conflict groups in $K(r_1, r_2)$ for the databases r_1 and r_2 in Example 6-2, namely $\kappa_1 = (A_2, 2)$ and $\kappa_2 = (A_4, 3)$. Note that the set $K(r_2, r_1)$ for the same databases contains three conflict groups, i.e., $\kappa_1 = (A_2, 1)$, $\kappa_2 = (A_4, 0)$, and $\kappa_3 = (A_4, 1)$. Due to the definition of the modification operations, we need at least one modification operation for solving the conflicts represented by a given conflict group.

Lemma 6.3: The lower bound for the update distance between a pair of databases r_1 and r_2, denoted by $LB(r_1, r_2)$, is given by:

$$LB(r_1, r_2) = |U(r_2, r_1)| + |K(r_1, r_2)| + \begin{cases} 1, if\ U(r_1, r_2) \neq \{\} \\ 0, else \end{cases}$$

Proof: In order to transform r_1 into r_2 with the restrictions for update operations as described above, we need exactly one insert operation for each tuple in $|U(r_2, r_1)|$, at least one modification operation for each conflict group in $K(r_1, r_2)$, and at least one delete operations if there are tuples to be deleted. ■

For the example in Figure 6-2 the update distance is three, as shown by the update sequence in c). The lower bound of the update distance is one and the upper bound is five. For the databases in Example 6-2 the update distance $\Delta_U(r_1, r_2)$ is two, which is also the lower bound.

6.2 TRANSIT - Minimal Transformers for Databases

This section describes the TRANSIT algorithms which determine the set of minimal transformers for contradicting databases. Regarding the minimization of transformer length, an early execution of insert operations does not provide any benefit. Instead, early inserts bear the chance that following modification or delete operations affect the inserted tuples and cause additional contradictions. Inserts are therefore delayed until all other contradictions have been eliminated. Delete operations can be handled as special cases of conflict resolution with modification operations. We therefore omit the separated treatment of deletes and postpone this to Section 6.2.4. In the following, we only consider modification operations and restrict the algorithms to databases pairs without unmatched tuples.

Given a pair of databases r_o and r_t, called *origin* and *target*, the TRANSIT algorithms enumerate the space of all databases reachable by applying sequences of modification operations to r_o. Doing so efficiently poses several challenges for which we describe solutions. First, we introduce transition graphs as formalizations of the search problem. Since many update sequences lead to the same database state, duplicate detection is of outermost importance. We describe a hashing scheme for efficient duplicate checking. We show how we use upper and lower bounds defined in Section 6.2.2 to prune the search space leading to a branch and bound algorithm. We then describe a breadth-first strategy for traversing the search space and briefly sketch a depth-first strategy. In Section 6.2.3, we show how – given a database state – the set of all possible modification operations can be computed using an algorithm that computes closed frequent itemsets. Finally, Section 6.2.4 explains how to handle delete operations as a special case of modification operations and briefly discusses the usage of unrestricted sequences of update operation. Throughout all other sections of this thesis, however, we limit all explanations to modification operations and ignore delete and insert operations.

6.2.1 Search Space Exploration

Given a pair of databases r_o and r_t our goal is to determine $T(r_o, r_t)$. Our approach starts by determining all database states derivable from r_o by a single modification operation. We call the resulting databases *level-1* databases. *Level-2* databases are computed by using all *level-1* databases as starting point for another modification. This process continues until we reach r_t. The level at which the target is reached first reflects the update distance $\Delta_U(r_o, r_t)$. To determine $T(r_o, r_t)$ the algorithm also needs to enumerate all other sequences that are of the same length. We maintain the sequence of modification operations with each database. Since different update sequences may generate the same database, databases generated at level n may have an update distance that is actually shorter than n. We later treat the detection of duplicated databases. Since we enumerate all possible modifications at each level and for each database, we ensure that our first match with r_t defines the shortest possible sequence.

Transition Graph

We represent the search space using a directed labeled graph, called transition graph. Vertices of this graph are databases connected by directed edges representing modification operations.

Definition 6.12: For two databases r_o and r_t, the *transition graph* $G_T = (V, E)$ with vertices V and edges E is defined as follows: V is the set of all databases derivable from r_o using an update sequence of length shorter than or equal to the update distance $\Delta_U(r_o, r_t)$. This implies that $r_t \in V$. E is the set of all edges $e = (r_1, r_2, \psi)$ for which $\psi(r_1) = r_2$, $r_1, r_2 \in V$. ■

The update operation represents the edge label, denoted by $label(e)$. We call r_1 the source of e, denoted by $source(e)$, and r_2 the target of e, denoted by $target(e)$. A path between two databases r_1 and r_2 within the transition graph is a sequence of edges that connect r_1 with r_2.

Definition 6.13: A *path* $\varphi = \langle e_1, \ldots, e_p \rangle$ within transition graph $G_T = (V, E)$ is a sequence of edges from E with $source(e_i) = target(e_{i-1})$ for all $1 < i \leq p$. ■

Two databases r_1 and r_2 are *connected* by φ if $source(e_1) = r_1$ and $target(e_p) = r_2$. Each path between two databases r_1 and r_2 defines a transformer for r_1 and r_2. The path $\varphi = \langle e_1, \ldots, e_p \rangle$ represents a transformer $\Psi = \langle label(e_1), \ldots, label(e_p) \rangle$, with $\Psi(source(e_1)) = target(e_p)$. A path is *minimal* if no shorter path between the same two databases exists. Clearly, the set of minimal transformers for r_o and r_t is given by all minimal paths from r_o to r_t within the transition graph. For a transition graph $G_T = (V, E)$ the *minimal transition graph* $G_{T_{min}} = (V_{min}, E_{min})$ with $V_{min} \subseteq V$ and $E_{min} \subseteq E$ is the part of the transition graph G_T containing only vertices and edges that are contained in the minimal paths between r_o and r_t.

The TRANSIT algorithms iteratively construct the transition graph – or a part of it containing at least the minimal transition graph – starting with r_o as the only vertex. Figure 6-3 shows an example of such a transition graph. The different levels are outlined by horizontal lines and derivable databases are only shown at the level of their update distance from r_o (as opposed to showing them on each level at which they are derived). Vertices and edges of the minimal transition graph are enclosed within a grey box.

Duplicate databases while constructing the transition graph occur whenever the same database is derived by different update sequences. We distinguish between *inter-level* and *intra-level* duplicates. Inter-level duplicates occur, if update sequences of different length derive the same database, i.e., the same databases is derived at different levels. Duplicates at different levels of the graph may introduce cycles. Since the corresponding edges – delineated by dotted lines for clarity in Figure 6-3 – cannot be part of a minimal transformer, they are not included in the graph. This approach ensures that the resulting graph is acyclic. Intra-level duplicates result from different update sequences of equal length that derive the same database. These duplicate databases result in multiple edges between two vertices on adjacent distance levels.

Duplicate Detection

A large portion of databases generated in transition graph enumeration are duplicates due to different update sequences deriving the same database. For example, the operations $\psi_1 = ((A_3, 0), \{(A_3, 1)\})$ and $\psi_2 = ((A_3, 0), \{\})$ derive the same result when applied to database r_1 of Figure 6-2. Also, may update sequences derive the same database from itself. For example, the update sequence $\langle((A_2, 0), \{(A_1, 1)\}), ((A_2, 1), \{(A_1, 1)\})\rangle$ derives r_1 from r_1 using a 2-step update sequence. We must detect duplicates efficiently to avoid unnecessary explosion of the search space.

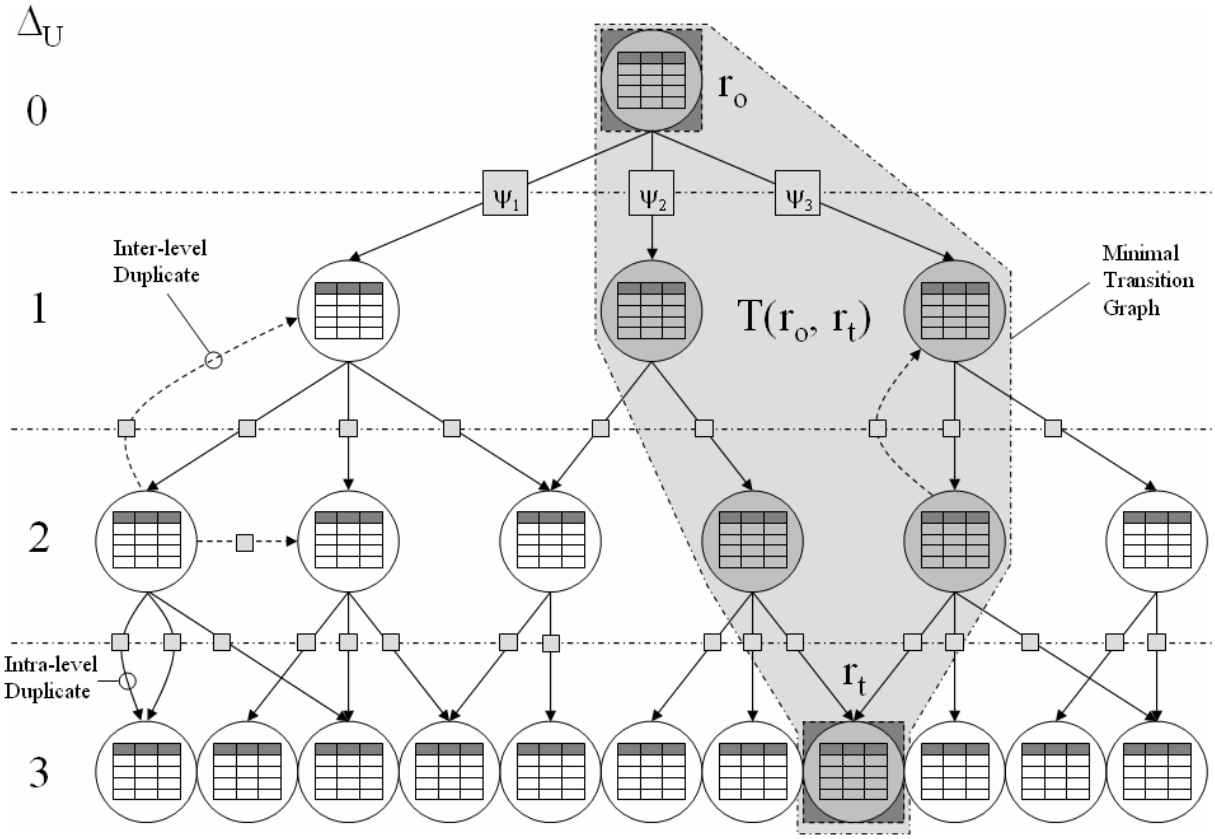


Figure 6-3: An exemplified transition graph as generated by the TRANSIT algorithm without pruning.

Example 6-3: Figure 6-4 shows a pair of databases having an update distance $\Delta_U(r_1, r_2) = 4$. In the table below, we list at each level of the transition graph the number of newly generated databases, the number of duplicates, and the overall number of executed modification operations. The table shows the exponential growth in generated databases (from 111 at level 1 to 4,625,519 at level 4) as well as the correspondingly increasing number of executed modification operations. There are a total of 24,586,604 executed modification operations while generating the derivable databases. The final transition graph has a total of 4,823,538 vertices and 16,997,183 edges. The table also shows, that there is a very high rate of generated duplicates (ca. 80% of the generated databases), thus necessitating and justifying additional effort in order to detect and remove these duplicates. This number would even be higher if duplicate databases where not detected and removed from the graph.

	Generated Databases	Inter-level Duplicates	Intra-level Duplicates	Operations executed
$\Delta_U = 1$	111	0	11	122
$\Delta_U = 2$	5,761	1,905	5,585	13,251
$\Delta_U = 3$	192,146	165,303	340,871	698,320
$\Delta_U = 4$	4,625,519	7,422,214	11,827,178	23,874,911
$\sum$	4,823,537	7,589,422	12,173,645	24,586,604

■

r1

A_1	A_2	A_3	A_4
1	1	2	3
2	1	3	3
3	1	2	1
4	1	2	2
5	1	2	7
6	1	2	6
7	2	2	5
8	0	2	6

r2

A_1	A_2	A_3	A_4
1	2	2	6
2	2	3	6
3	2	3	1
4	2	3	2
5	2	3	7
6	2	3	6
7	2	2	5
8	0	2	6

Figure 6-4: A pair of databases having an update distance $\Delta_U(r_1, r_2)$ of four.

Duplicate detection requires comparison of entire databases, i.e., the complete scan of two databases. To reduce the number of duplicate checks, we compute a hash value for each database and maintain a hash table for generated databases. Complete database comparisons are only performed when the hash values of two databases are equal, which drastically reduces the number of (expensive) full database comparisons at the price of having to maintain the hash table. We currently employ the following hash function for databases: Without a loss of generality we assume the primary keys to be integers in the range *1*, …, *m*. We number the attribute values of the particular tuples in the following order $0{:}t\{1\}[A_1]$, $1{:}t\{1\}[A_2]$, …, $(n * m) - 1{:}t\{m\}[A_n]$, called the cell index. With each database we maintain a list of the conflicting values with an order based on this cell index (Figure 6-5 shows such a list for the conflicts between databases r_1 and r_2 in Figure 6-4). Starting at position *0*, we select *k* values from this list, having cell index positions $c_1, \ldots, c_k$, with $c_i = (i - 1) * (\textit{number of conflicts} / k)$, for $1 \leq i \leq k$. The final hash value is an integer with *k* digits, where the i^{th}-digit is the value of cell c_i *modulo 10*. For the example in Figure 6-5 for $k = 4$ we use the values at position *0*, *3*, *6*, and *9* with the resulting hash value being *2131* (the position of the digits being numbered from right to left).

We also tested a hash function based on a histogram of the attribute values occurring within a database. We thereby maintain for each occurring value from $dom(A)$ the number of its occurrences within the databases (also shown in Figure 6-5). From this list we again take *k* values to compute the hash value. We found the later scheme to be inferior in our experiments.

Representation of Databases

The usage of a hash table greatly reduces the number of comparisons of complete databases. Still, in cases of equal hash values we must access the actual values in the database for comparison, i.e., we need access to the complete database in the vertices. We implemented two different representations of these databases. In a first representation we maintain complete databases within the vertices. While this is memory consuming the access to the actual data values is fast. Alternatively, for each database *r* only a transformer $\Psi(r_o) = r$ is maintained within the corresponding vertex. This greatly reduces the memory requirement for transition graph maintenance. As a downside, we are now forced to re-derive the database *r* from the origin every time access to the actual tuples and attribute values is required. Therefore, this representation even further depends on the ability of the hash function to generate equally distributed hash values for the databases in order to avoid collisions.

r_1

A_1	A_2	A_3	A_4
1	1	2	3
2	1	3	3
3	1	2	1
4	1	2	2
5	1	2	7
6	1	2	6
7	2	2	5
8	0	2	6

Cell Index

A_1	A_2	A_3	A_4
1	*2*	*3*	*4*
5	*6*	*7*	*8*
9	*10*	*11*	*12*
13	*14*	*15*	*16*
17	*18*	*19*	*20*
21	*22*	*23*	*24*
25	*26*	*27*	*28*
29	*30*	*31*	*32*

List of Conflicting Values

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
Index	*2*	*4*	*6*	*8*	*10*	*11*	*14*	*15*	*18*	*19*	*22*	*23*
Value	1	3	1	3	1	2	1	2	1	2	1	2

Value Histogram

	0	*1*	*2*	*3*	*4*	*5*	*6*
Value	0	1	2	3	5	6	7
Occur.	1	7	9	3	1	2	1

Figure 6-5: Cell index and list of conflicting values as used for the hashing function.

6.2.2 Branch and Bound Algorithms

The TRANSIT algorithms try to avoid generating the complete transition graph. The number of vertices outside of the minimal transition graph in Example 6-3 shows that many of the generated databases are not part of any minimal transformer. This observation is supported by examining the minimal transition graph for the databases of Figure 6-4. The minimal transition graph contains 18 vertices and 36 edges. These numbers are far below the number of approximately 5,000,000 vertices and 17,000,000 edges in the constructed transition graph.

Pruning

The large difference between the number of vertices in the minimal transition graph and the total number of databases generated suggest that pruning is essential. In TRANSIT, pruning uses the upper and lower bounds for the update distance as defined in Lemma 6.2 and Lemma 6.3. Let β denote the current upper bound for the update distance between r_o and r_t. This bound is initialized as $UB(r_o, r_t)$ following Lemma 6.2. Each generated database r with $LB(r, r_t) > (\beta - \Delta_U(r_o, r))$ is not included in the transition graph, because any path from r_o to r_t through r will have at least $\Delta_U(r_o, r) + LB(r, r_t) > \beta$ edges and is therefore not minimal. The update distance $\Delta_U(r_o, r)$ is maintained with each vertex in order to avoid recalculation. We decrease β whenever a database r is generated having an upper bound below the currently best bound, i.e., $(\Delta_U(r_o, r) + \Delta_R(r, r_t)) < \beta$. For such a database there exists a transformer $\Psi(r_o) = r$ with length $|\Psi| = \Delta_U(r_o, r)$. Lemma 6.1 guarantees the existence of a transformer $\Psi'(r) = r_t$ with length $|\Psi'| = \Delta_R(r, r_t)$. The following simple lemma proves the existence of a transformer $\Psi''(r_o) = r_t$ having length $|\Psi''| = \Delta_U(r_o, r) + \Delta_R(r, r_t)$.

Lemma 6.4: Given transformers $\Psi_1(r_1) = r_2$ and $\Psi_2(r_2) = r_3$, there exists a transformer $\Psi_3(r_1) = r_3$ with length $|\Psi_3| = |\Psi_1| + |\Psi_2|$.

Proof: Transformer Ψ_3 is a concatenation of $\Psi_1 = <\psi_{11}, \ldots, \psi_{1k}>$ and $\Psi_2 = <\psi_{21}, \ldots, \psi_{2p}>$, i.e., $\Psi_3 = <\psi_{11}, \ldots, \psi_{2k}, \psi_{21}, \ldots, \psi_{2p}>$. The length of Ψ_3 is $|\Psi_1| + |\Psi_2|$ and the result of $\Psi_3(r_1)$ equals $\Psi_2(\Psi_1(r_1))$ which is r_3. ■

Each time the bound β is decreased we remove all databases from the transition graph with insufficient bound, i.e., for which $\Delta_U(r_o, r) + LB((r, r_t) > \beta$.

The described approach resembles a branch and bound behavior [LD60]. The branch and bound algorithm generates all databases derivable by update sequences of increasing length. Within the branch step one of the unprocessed databases is chosen for processing. We generate all databases that are derivable from this database by a single modification operation. Next, in the bound step the current bound is decreased if possible and databases are pruned as described. After finishing the processing of the current database we chose the next database for processing from the remaining, untested databases in the graph. We continue until r_t is reached and no untested database remains. In a branch and bound algorithm, we can explore the search space either in breadth-first or in depth-first manner. We describe a breadth-first algorithm first and a depth-first algorithm afterwards.

Breadth-First Algorithm

In the breadth-first algorithm, we process all databases at the current level first before proceeding to databases at the next level. Figure 6-6 shows the changes to the transition graph from Figure 6-3 when using a breadth-first approach. The fictitious upper and lower bounds of the databases are shown in white boxes on the right of every vertex. The order in which the databases are processed is given by

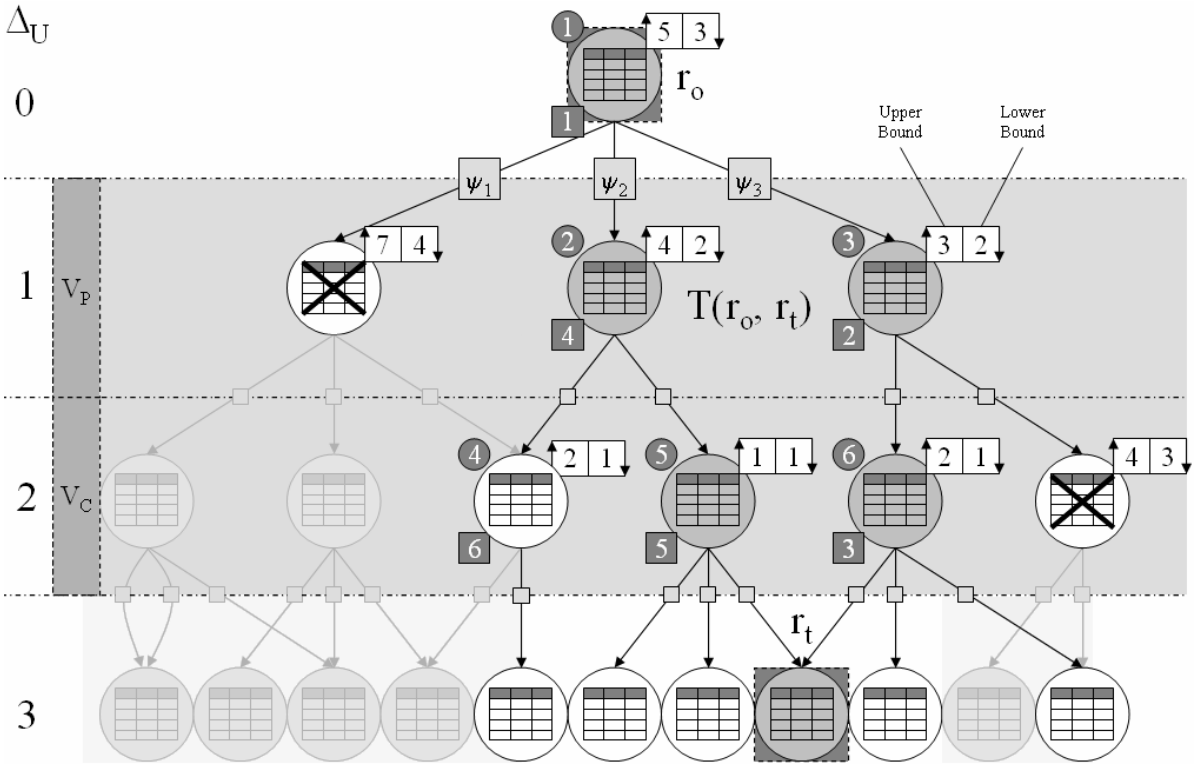


Figure 6-6: Exemplified transition graph construction in a branch and bound approach. The upper and lower bounds are shown in white boxes on the right of every vertex. Dark grey circles attached to the vertices reflect the order of database processing in a breadth-first approach. The order of database processing in a depth-first approach is reflected by the number in the dark grey squares.

the number in the dark grey circles attached to the left of the vertices. For example, the database on the left side of *level 1* is pruned, because every path from r_o to r_t through this vertex is at least of length 5, while the current bound is 4 after generating all databases at level 1. Due to the pruning of databases in the bound step large portions of the originally shown transition graph are not generated or tested in the breadth-first approach.

The corresponding algorithm TRANSIT-BFS is shown in Figure 6-7. Let V_P and V_C denote the set of databases from the previous and the current level of the transition graph G_T respectively. Variable Δ_U stores the depth of G_T. We initialize V_P with $\{r_o\}$. Each database in V_P is processed to enumerate V_C (*lines 9-26*). Databases in V_C then become the candidates for enumeration of the next level (*line 27*). We sort the candidates in ascending order of their lower and upper bounds. This is done with the intention of being able to decrease the current bound β as soon as possible and avoid unnecessary insertion of databases that are pruned afterwards. After reaching the destination the algorithm

```
1  TRANSIT-BFS(r_o, r_t) {
2    G_T := ({r_o}, {});
3    V_P := V(G_T);
4    Δ_U := 0;
5    β := Δ_R(r_o, r_t);
6    while(r_t ∉ V_P) {
7      Δ_U := Δ_U + 1;
8      V_C := {};
9      for each r_i ∈ V_P do {
10       for each ψ ∈ modifier(r_i, r_t) do {
11         r_new := ψ(clone(r_i));
12         if ((LB(r_new, r_t) + Δ_U) ≤ β) {
13           if (r_new ∉ V(G_T)) {
14             V(G_T) := V(G_T) ∪ {r_new};
15             E(G_T) := E(G_T) ∪ {(r_i, r_new, ψ)};
16             V_C := V_C ∪ {r_new};
17             if ((Δ_R(r_new, r_t) + Δ_U) < β) {
18               β := Δ_R(r_new, r_t) + Δ_U;
19               prune V_P, V_C, G_T, β;
20             }
21           } else if (r_new ∈ V_C) {
22             E(G_T) := E(G_T) ∪ {(r_i, r_new, ψ)};
23           }
24         }
25       }
26     }
27     V_P := sort(V_C);
28   }
29   output min_paths(G_T, r_o, r_t);
30 }
```

Figure 6-7: The breadth-first algorithm TRANSIT-BFS.

returns the set of minimal paths in the transition graph from the origin to the target (*line 29*). In Figure 6-6 the databases in the sets V_p and V_C are highlighted for the construction of *level 2*. If we are only interested in calculating the update distance, the algorithm can terminate immediately after r_t is derived for the first time (check for equality after *line 11*).

Processing a database r_i from V_P starts by determining the set of possible modification operations, denoted by *modifier*(r_i, r_t) (*line 10* – see Section 6.2.3 for details). Each operation is applied to a copy of the database, as modification operations alter the given database (*line 11*). The resulting database r_{new} is added to the transition graph if its lower bound with r_t does not exceed the current bound. If r_{new} does not occur within G_T it is added to V_C (*lines 12-24*). If r_{new} does occur within G_T, we have to check whether it is an intra-level duplicate or an inter-level duplicate (*lines 21-23*). In the former case the database has been derived before at the current distance level and we add an additional edge to G_T. In the latter case no changes occur.

Depth-First Algorithm

We refer to the algorithm that constructs the transition graph in depth-first manner as TRANSIT-DFS. Within this algorithm, after finishing the processing of the current database, i.e., generating all databases derivable with a single modification operation, we immediately proceed to the next distance level. From all generated database, we chose the one with the smallest lower bound as new current database. Pruning is performed as described above. The depth-first approach finds a first solution after processing fewer databases then the breadth-first approach. Although this solution is not necessarily minimal in the number of modification operations, it often helps to perform additional pruning. After reaching the target database, TRANSIT-DFS needs to return to the previous databases and process them as candidates, again in a depth-first manner. This is continued until all databases that have not been pruned by the bounding step have been tested. The order in which databases are processed in TRANSIT-DFS is reflected by the number in the dark grey squares attached to the bottom left of each vertex in Figure 6-6. In TRANSIT-DFS we maintain the databases processed and generated on the current path on a stack to enable upward traversal. Compared to TRANSIT-BFS detection of duplicate databases is complicated by the fact that identical databases may be generated multiple times at decreasing levels. Every time a database is repeatedly derived at a lower level, it has to be considered as a candidate again. Due to the depth-first proceeding we temporarily add databases to the transition graph having an update distance from r_o above the update distance $\Delta_U(r_o, r_t)$. This is contrary to our definition of the transition graph, where we only consider databases having an update distance below or equal to $\Delta_U(r_o, r_t)$. However, due to the performed pruning, we will remove any databases r with an update distance $\Delta_U(r_o, r) > \Delta_U(r_o, r_t)$ from the final transition graph.

Example 6-4: The advantage of using the branch and bound approaches is shown by comparing the following numbers with those in Example 6-3. Using TRANSIT-BFS, the minimal transformers for the two databases r_1 and r_2 in Figure 6-4 is found after exploring only 255 databases (compared to 198,019 databases that are processed in total on the first three levels listed in Example 6-3). TRANSIT-BFS executes a total of 42,010 modification operations (compared to 24,586,604) and generates 3,651 databases. For the depth-first approach the number are even lower. TRANSIT-DFS processes a total of 1,433 databases, with 226,655 modification operations executed and 1,609 databases generated. ■

6.2.3 Enumeration of Valid Modification Operations

Following Definition 6.3, a modification operation is a pair consisting of modification term τ and modification pattern ρ. We are only interested in enumerating valid modification operations that change the database (Definition 6.9). For a database r the set of valid modification operations is a subset of the Cartesian product of the set of modification terms and the set of modification patterns.

The Set of Modification Terms

Terms are attribute-value pairs. Within modification terms only non-key attributes are permitted. The set of valid modification terms is the union of valid modification terms for each non-key attribute. For each attribute $A \in R \setminus ID$ this set is $A \times dom(A)$. A problem is the infinite size of $dom(A)$ that leads to an infinite set of modification terms and therefore an infinite set of modification operations. Consequently, the algorithm would not terminate, although almost all of the generated databases are isomorphic with respect to their ability to participate in a shortest update sequence. We therefore constrain the set of possible modification values. We accomplish this goal by using the values occurring within the current database r and the target r_t. In summary, we permit the following values for modification terms for attribute A:

- All values from the target that occur within attribute A, denoted by $r_t[A]$. Some of these values have to be used at least once as modification value for conflict solution. The values that are not used for conflict solution are also contained in the following set.
- All values occurring for attribute A in the current database r, denoted by $r[A]$. In some situations increasing the selectivity of individual values enables to solve more conflicts using a single modification operation afterwards. An example is shown in Figure 6-8. Without allowing existing values to be used as modification values, we need at least seven operations for solving the existing conflicts. If we set the values in attribute A_5 for tuples $t\{7\}$ and $t\{8\}$ to *1* instead, we are enabled to solve the conflicts in attributes A_2 to A_5 using a single operation afterwards, resulting in a total of five modification operations.
- Any of the remaining values from $dom(A)$ not contained in $r_t[A] \cup r[A]$ is a potentially necessary modification value, possibly to serve as a unique selection criterion in later stages of the algorithm. Thus, the actual value does not matter, as long as it is different from all other currently used values. We chose one value using a random function. We call these values *Skolem constants*.[4]

The Skolem constants are maintained within a separate list for each attribute, called $skolem(A)$. Within the final modification sequence, the occurring Skolem constants can be replaced by any valid subset of $dom(A)$ of size $|skolem(A)|$ that is disjoint with $r_o[A] \cup r_t[A]$.

The Set of Selection Patterns

For every subset of tuples in a database that is selectable by a pattern ρ there has to be a modification operation that allows modification of these tuples. Let $P(r)$ denote the set of patterns that select at least one tuple from r. This set very likely contains redundant pairs of patterns ρ_1, ρ_2 with $\rho_1 \neq \rho_2$ and

[4] The numbers in Example 6-3 result from experiments without using Skolem constants. If Skolem constants are used the number of operations executed (and databases generated) for the first three levels increase to 164 (153), 25,051 (11,755), and 1,989,604 (624,659).

$\rho_1(r) = \rho_2(r)$. Enumerating modification operations using the patterns in $P(r)$ would result in operations with equal effect. Therefore, we restrict the set of modification patterns to the set of closed patterns as defined in Definition 4.7. Let $P_C(r)$ denote the set of closed patterns that select at least one tuple from r. A closed pattern ρ represents exactly those terms that occur within every tuple of $\rho(r)$ and there are no two patterns $\rho_1, \rho_2 \in P_C(r)$ with $\rho_1 \neq \rho_2$, that select equal subsets of r. The following lemma proves that for each pattern $\rho \in P(r)$ there exists exactly one corresponding closed pattern $cp(\rho)$ in $P_C(r)$ (see Definition 4.8).

Lemma 6.5: Given a database r. For each pattern $\rho \in P(r)$ there exists a pattern $\rho' \in P_C(r)$ with $\rho(r) = \rho'(r)$.

Proof: Following our definitions in Section 5.3 $pattern(\rho(r))$ denotes the set of terms common to the tuples in $\rho(r)$. This set forms a closed pattern for $\rho(r)$. All tuples in $\rho(r)$ satisfy $pattern(\rho(r))$ and if we add a term to $pattern(\rho(r))$ the pattern will no longer be satisfied by all tuples in $\rho(r)$. Therefore, $pattern(\rho(r))$ equals $\rho' \in P_C(r)$ with $\rho(r) = \rho'(r)$. ■

Based on Lemma 6.5 it is sufficient to use $P_C(r)$ extended by the empty pattern instead of $P(r)$ as the set of valid modification patterns. We add the empty pattern to $P_C(r)$ in order to allow modifications of the complete database at once. To determine the set of closed patterns, we start with a single scan of the database. Each tuple is a closed pattern due to the primary key constraint. While scanning the database, we determine the set of terms for each attribute and maintain a list of tuples, in which these terms occur. We then prune all terms having a support, i.e., a tuple list size, of 1. These tuples only occur in the closed patterns already represented by single tuples. We then use CHARM [ZH02] to enumerate the set of closed patterns.

r_1

A_1	A_2	A_3	A_4	A_5
1	10	18	26	1
2	11	19	27	1
3	12	20	28	1
4	13	21	29	0
5	14	22	30	0
6	2	23	31	34
7	15	3	32	35
8	16	24	4	36
9	17	25	33	5

r_2

A_1	A_2	A_3	A_4	A_5
4	2	3	4	5
5	2	3	4	5
6	2	3	4	5
7	2	3	4	5
8	2	3	4	5
9	2	23	31	34
2	15	3	32	35
3	16	24	4	36
10	17	25	33	5

Figure 6-8: An example for enhancing the selectivity of existing patterns in order to find an optimal solution. We need two modification operations to set the values in $t\{7\}[A_5]$ and $t\{8\}[A_5]$ to 1. We can then solve all conflicts using four modification operations. Without increasing the set of tuples selected by term $(A_5, 1)$ we need at least 7 operations, e.g. two operations to solve the conflicts in A_5, one operation to change the value of $t\{9\}[A_5]$, three operations to solve the conflicts in attributes A_2, A_3, and A_4, and one operation to set the value of $t\{9\}[A_5]$ back to 5.

Filtering Valid Modification Operations

A modification operation has no effect if the modification term τ also occurs within the modification pattern. In this case, all selected tuples already possess the new value in the modified attribute. We remove these operations. In Figure 6-9, the set of valid modification operations for database r_1 is 65. There are 8 closed patterns (including the empty pattern). If we assume $skolem(A_2) = \{9\}$, $skolem(A_3) = \{9\}$, and $skolem(A_4) = \{9\}$, there are 10 modification terms. As a result we receive a total of 65 valid modification operations. The number shows that even for very small databases the number of valid modification operations can be large.

r_1

A_1	A_2	A_3	A_4
1	2	1	1
2	1	2	1
3	1	2	0
4	1	2	1

r_2

A_1	A_2	A_3	A_4
1	2	1	3
2	2	2	3
3	2	2	3
4	2	2	3

Patterns	Terms
$\rho_1 = \{(A_1, 1), (A_2, 2), (A_3, 1), (A_4, 1)\}$ $\rho_2 = \{(A_1, 2), (A_2, 1), (A_3, 2), (A_4, 1)\}$ $\rho_3 = \{(A_1, 3), (A_2, 1), (A_3, 2), (A_4, 0)\}$ $\rho_4 = \{(A_1, 4), (A_2, 1), (A_3, 2), (A_4, 1)\}$ $\rho_5 = \{(A_2, 1), (A_3, 2)\}$ $\rho_6 = \{(A_4, 1)\}$ $\rho_7 = \{(A_2, 1), (A_3, 2), (A_4, 1)\}$ $\rho_8 = \{\}$	$(A_2, 1), (A_2, 2), (A_2, 9),$ $(A_3, 1),(A_3, 2),(A_3, 9),$ $(A_4, 0),(A_4, 1),(A_4, 3),(A_4, 9)$

Figure 6-9: The set of modification patterns and terms for database r_1 resulting in a total of 65 valid modification operations.

6.2.4 Handling Delete and Insert Operations

To conclude the discussion of the TRANSIT algorithms, we now describe the handling of insert and delete operations. The execution of necessary insert operations is performed after solving existing conflicts. Delete operations are handled as special cases of modification operations. We therefore need to slightly alter a given database. The set of tuples from r_o to be deleted is given by $U(r_o, r_t)$. We add a special attribute A_D to schema R setting $t[A_D] = 0$ for each $t \in U(r_o, r_t)$. We also insert the tuples from $U(r_o, r_t)$ to r_t changing the value of $t[A_D]$ to 1. For all other tuples in databases r_o and r_t attribute A_D is undefined. The attribute A_D acts as a delete flag and the values are altered using modification operations. Terms for attribute A_D are excluded when enumerating valid selection patterns. However, A_D is allowed as attribute in modification terms. The only valid modification value in these terms is 1. A modification operation $\psi_\mu = (\tau, \rho)$ with $\tau = (A_D, 1)$ represents a delete operation $\psi_\delta = (\rho)$. A tuple t with $t[A_D] = 1$ then represents a deleted tuple. Following the Definition 6.9 of valid update operations, we additionally have to restrict ρ to select only tuples from $U(r_o, r_t)$. In the resulting modification sequences the modification operations with $\tau = (A_D, 1)$ are replaced by the appropriate delete operations.

The described algorithm for enumerating valid modification operations is easily extended to allow unrestricted insert and delete operations within update sequences, i.e., ignoring Definition 6.9 except that the update operation has to change the database. When enumerating modification operations for a database r we (i) add an insert operation for every tuple in r_t / r, and (ii) add a delete operation for every valid modification pattern.

Upper and Lower Bounds using Insert and Delete

The TRANSIT algorithms operate independently from the set of allowed update operations. Allowing insert and delete operations, enables the application of TRANSIT on database pairs that do not completely overlap. However, including insert and delete implies changes to the definitions for the upper and lower bounds in Lemma 6.2 and Lemma 6.3. For every pair of databases r_1 and r_2 there is always a trivial transformer $\Psi(r_1) = r_2$ that (i) deletes all tuples from r_1 and (ii) successively insert all tuples from r_2.

Lemma 6.6: For update sequences allowing unrestricted operations $UB(r_1, r_2) = min(\Delta_R(r_1, r_2), |r_2| + 1)$ provides an upper bound.

Proof: According to Lemma 6.2 there exists a transformer of length $\Delta_R(r_1, r_2)$. However, we now have to consider that the trivial transformer as described above is minimal. The trivial transformer for r_1 and r_2 is given by $|r_2| + 1$, i.e., one delete operation and $|r_2|$ insert operations. ■

Lemma 6.7: For update sequences allowing unrestricted operations the lower bound is defined by:

$$LB(r_1, r_2) = min(|U(r_2, r_1)| + |K(r_1, r_2)| + \begin{cases} 1, if\ U(r_1, r_2) \neq \{\} \\ 0, else \end{cases}, |r_2| + 1).$$

Proof: Similar to the upper bound, we now need to consider that the number of tuples in r_2 may be lower than the number of conflict groups in $K(r_1, r_2)$. ■

6.3 Experimental Results

We now discuss results of our experiments using the described algorithms on different pairs of databases. We use a set of four small databases pairs in these experiments, named DBP1-DBP4 in the following. Three of these pairs are shown in this thesis. DBP1 corresponds to the databases in Figure 6-2, DBP2 to Figure 6-4, and DBP3 to the database in Example 7-2. DBP4 is the pair of databases named F1 in [MLF06b]. We also performed experiments on larger databases in conjunction with the heuristics described in the following chapter. The necessary effort to determine the set of minimal transformers for the four pairs of databases is shown in Figure 6-10 a). In the first two columns the number of databases processed and modification operations executed for building the transition graph are shown. Also listed are the overall number of databases added to the graph and the number of databases generated as duplicates. The final results, i.e., the size of the minimal transition graphs, the total number of minimal transformers (|T|), and the update distances (Δ_U), are shown in Figure 6-10 c). Figure 6-10 b) shows the number of valid modification operations that were executed when processing the origin database of all four database pairs. Note that all experiments determined only the set $T(r_1, r_2)$ of minimal transformers. Also shown in Figure 6-10 b) are the number of generated databases and edges when processing the origin database of the four database pairs.

a)

TRANSIT-BFS	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	OT[1]	TG[2]	OG[3]
DBP1	279	38,006	3,026	2,832	968	136.22	0.09	12.56
DBP2	255	42,010	3,651	3,481	749	164.75	0.07	11.51
DBP3	32,695	12,524,800	317,076	686,454	269,075	383.08	0.1	39.5
DBP4	12,742	5,457,202	89,424	159,695	42,421	428.28	0.14	61.03

TRANSIT-DFS	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	OT	TG	OG
DBP1	4,275	603,971	4,204	4,417	4,483	141.28	1.02	143.67
DBP2	1,433	226,655	1,609	1,625	871	158.17	0.89	140.87
DBP3	5,131	1,909,040	6,055	7,238	6,535	372.06	0.85	315.28
DBP4	95	36,986	1,134	373	32	389.33	0.08	32.62

[1] OT: The average number of operations executed per tested database
[2] TG: Percentage of tested databases from those added to the graph
[3] OG: The average number of operations executed per added database in the graph

b)

	Valid Operations	Databases Added	Edges Generated
DBP1	124	106	110
DBP2	164	149	159
DBP3	386	320	386
DBP4	451	398	441

c)

	Vertices	Edges	|T|	Δ_U
F4	6	6	2	3
F7	18	36	30	4
F14	30	76	160	5
F1	84	288	1,500	6

Figure 6-10: In a) the effort in number of modification operations executed and databases added to the transition graph by TRANSIT-BFS and TRANSIT-DFS are shown for different pairs of databases. In b) the effort for processing the origin database of all four database pairs is shown. The size of the minimal transition graphs, the total number of minimal transformers (|T|), and the update distances (Δ_U) are shown in c).

The number of executed modification operations is directly related to the number of tested databases. Comparing the average number of operations per tested database, shown in column OT of Figure 6-10 a), with the number of valid operations for each of the origins of the four database pairs (shown in Figure 6-10 b)) reveals, that this number remains quite similar for each of the generated and tested databases. This continuity implies an exponential growth of the number of executed modification operations if no pruning is performed. The large number of valid modification operations even for these small databases suggest that for larger databases the number of valid operations is going to explode.

For each databases tested the complete set of databases derivable by a single modification operation is generated. Each of these resulting databases is classified into one of four classes:

- **Rejected**: The resulting database has a lower bound that disqualifies it as an intermediate state of any minimal transformer. These databases are excluded from further consideration.
- **Newly Added**: The database has a sufficient lower bound and is added to the evolving transition graph.
- **Inter-Duplicate**: The database has been generated before at a lower distance level. The database therefore represents an inter-level duplicate and no changes to the graph occur.
- **Intra-Duplicate**: The database has already been derived at the current distance level, i.e., it is an intra-level duplicate. Intra-level duplicates cause the generation of an additional edge within the graph.

The large difference between the number of executed modification operations and the number of newly added and duplicate databases reveals that the majority of generated databases resulting from executed modification operations are pruned. The distribution of the generated databases on the four classes is exemplarily shown in Figure 6-11 for database pair DBP3. The distributions for the other database pairs in our experiments are fairly similar to the shown example. Figure 6-11 indicates that by far the largest portion of executed modification operations results in databases that are rejected. This portion is up to 90% when using the breadth-first approach, and even up to 99% for the depth-first approach. Figure 6-11 also indicates that a great portion of databases that are not rejected are duplicates. The large number of nearly 70% duplicates for the remaining databases justifies the effort for detecting and removing duplicates while constructing the transition graph.

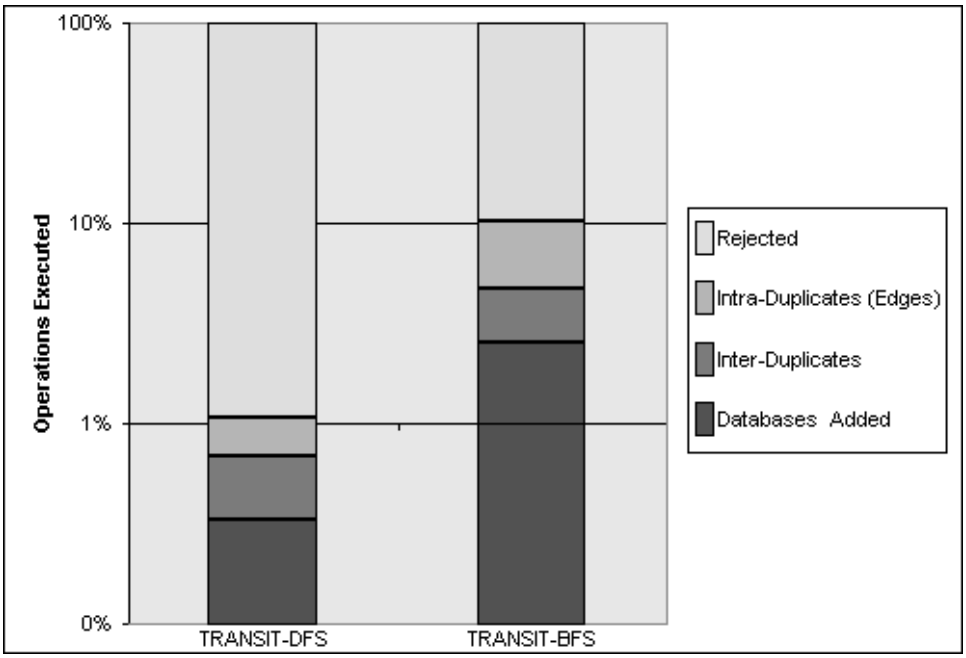


Figure 6-11: Results of modification operations are classified in one of four classes. Only a small fraction of the generated databases are actually added to transition graph while the majority is rejected due to insufficient lower bounds or as inter-level duplicates.

Figure 6-12 compares the number of executed modification operations (on the x-axis) with the number of databases resulting from these operations that are actually added to the transition graph (on the y-axis) for TRANSIT-DFS and DBP3. The advantage of pruning is clearly shown by the large discrepancy in the numbers. The total number of databases added to the transition graph is far below the number of executed modification operations. The linear growth of the number of databases added to the transition graph indicates that the number of added databases is approximately equal for all tested databases. However, pruning of databases once added to the graph is not very effective as indicated by the linear growth of the number of databases in the transition graph. This number increases linearly with the number of added databases. Thus, despite our quite effective pruning the transition graph remains large. The limits of pruning become even more obvious when comparing the large number of databases in Figure 6-12 with the number of databases in the final transition graph in Figure 6-10 c). As a result, the memory requirement of the transition graph is very large and it becomes impossible even for database pairs of mediocre size to maintain the graph completely in main memory.

Comparing the breadth-first and depth-first approaches shows that each of them has their strength and weaknesses. The depth-first approach is inferior for DBP1, where the optimal solution requires the insertion of conflicts at first. In all other cases, the depth-first approach performs better than the breadth-first approach with respect to the number of databases added and tested. The ratio of these numbers shown in column TG of Figure 6-10 a) indicates that the depth-first approach usually processes over 80% of the added databases as candidates at the next distance level, while the breadth-first approach adds numerous databases to the transition graph that are never considered as candidates afterwards. A special case is the number of added and tested databases by the depth-first approach for DBP1, where the ration is above 1. This discrepancy is due to those databases that are added once but tested several times at different (decreasing) distance levels.

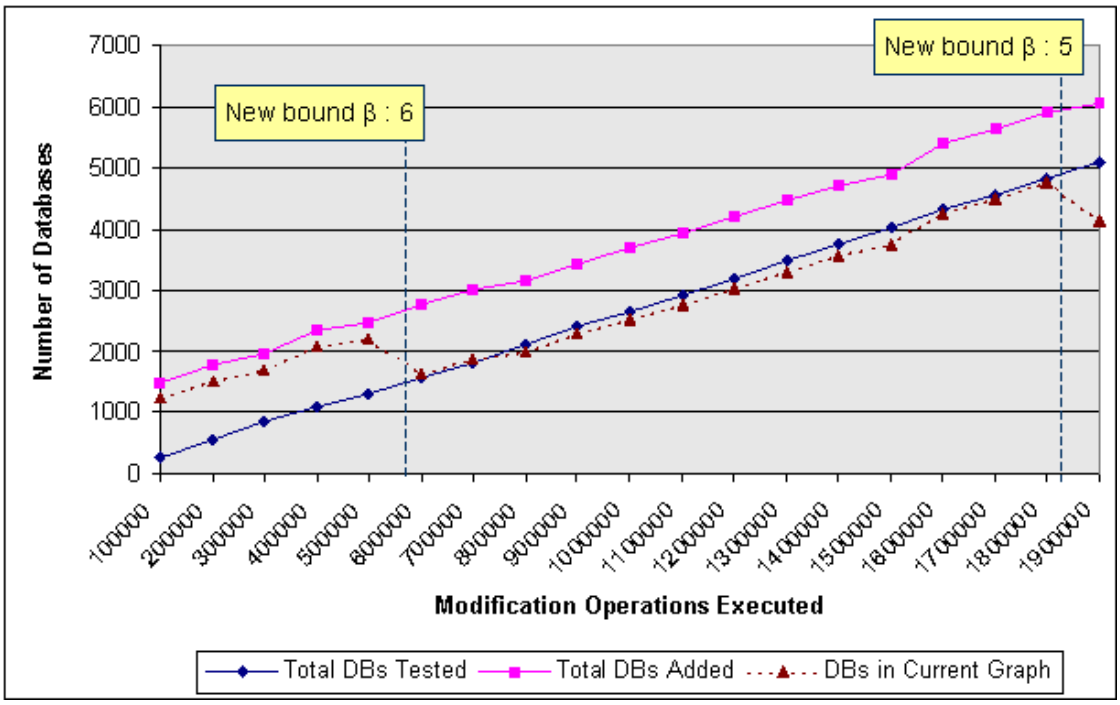


Figure 6-12: Development of the transition graph for TRANSIT-DFS on dataset DBP3.

Figure 6-13 shows the general advantage that TRANSIT-DFS has over TRANSIF-BFS. We list the number of added and tested databases for both approaches at each distance level for DBP3 in Figure 6-13. The breadth-first approach tends to peak at lower distance levels due to the limited pruning ability of the lower bound at earlier stages of processing. On the other hand, due to finding a first solution at an early stage, the depth-first approach has a better ability of pruning databases at the lower distance levels. Still, the number of databases in the generated transition graph is far above the number of databases in the minimal transition graph. This observation holds for all experimental dataset as comparison of the respective values in Figure 6-10 a) and Figure 6-10c) shows.

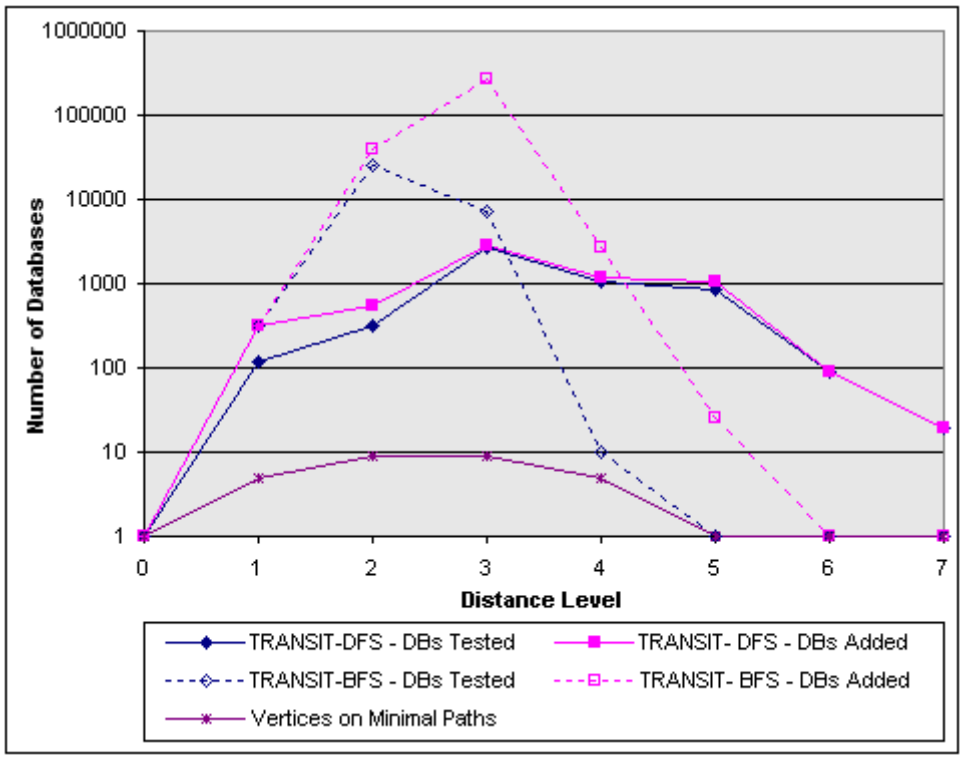


Figure 6-13: Comparing the exact algorithms for dataset DBP3.

6.4 Summary and Related Work

To the best of our knowledge the problem of finding minimal sequences of set-oriented operations for relational databases has not been considered before. There exist various distance measures for other objects, like the well-known *Hamming distance* [Ham50] or the *Levenshtein distance* [Lev65] for binary codes and strings. Our update distance follows the Levenshtein distance, defined as the minimal number of edit operations necessary to transform one string into another. We described a branch and bound approach for determining update sequences of SQL-like update operations that transform one database into another one. If conflicts between two databases are due to systematic manipulation, the operations within update sequences are valuable to domain experts interested in solving the conflicts. The only other distance measure for databases, which is related to our definition, is defined in [ABC99]. Here, the distance of two databases is defined as the number of tuples from each of the databases without a matching partner in the other database. This definition coincides with our definition of the resolution distance when disregarding existing conflicts and regarding only the existing uncer-

tainties. This definition is used in the area of computing consistent query answers for inconsistent databases [ABC99, CM05, Wij03]. The problem here is, given a query *Q*, a set of integrity constraints *IC*, and a database *r*, which violates *IC*, determine the set of tuples that satisfy *Q* and are contained in each possible repair for database r. A repair for database *r* is defined as a database *r'*, which satisfies *IC* and is minimal in distance to *r* in the class of all databases satisfying *IC* [ABC99]. While the approaches [ABC99, CM05] only allow insertion and deletion of tuples in order to find the repairs, [Wij03] also considers the modification of existing values. Opposed to these approaches, we do not rely on integrity constraints for the identification of contradicting values. Instead, in our model the repair is already given by the target database. We therefore are not interested in finding the nearest database in a plethora of possible repairs for an inconsistent database, but in identifying update sequences that transform a given database into another given database.

The manipulation of existing database values to satisfy a given set of integrity constraints is also considered in [BFFR05]. In this approach modification as well as insertion of tuples is allowed. A certain cost is assigned with each modification and insertion operation. For a given database and a set of integrity constraints, which are violated by the database, the problem then is to find a repair, i.e., a database satisfying a given set of constraints, with minimal cost. Again, in our approach we are not interested in determining the optimal value modifications in order to solve a set of conflicts, as the solutions of existing conflicts are predetermined by the target database. Our focus is rather on how to perform the (a priori known) necessary modifications with minimal effort in terms of the number of SQL-like update operations. All other approaches described so far do not consider this problem, as they implicitly expect to modify the values one at a time after they determined a conflict solution.

In [FLMC01], the authors discern between context dependent and context independent conflicts. Context dependent conflicts represent systematic disparities, which are consequences of conflicting assumptions or interpretations. Context independent conflicts are idiosyncratic in nature and are consequences of random events, human errors, or imperfect instrumentation. In this sense, we are considering context dependent conflicts. However, in contrast to [FLMC01], we do not consider complex data conversion rules for conflict resolution, but always use one of the conflicting values as the solution. Discovering conflict conversion rules is considered as future work. On the other hand, we do consider the conflict causing context to be identifiable as data patterns. Therefore, this work is a continuation of our work on mining patterns in contradictory data.

So called “update deltas” are used in several applications to represent differences between databases. In database versioning they are used as memory effective representation of different database version [DLW84]. However, versioning collects the actual operations during execution instead of having to reengineer them from two given versions. In [LG96] sequences of insert, delete and update operations are used to represent differences between database snapshots. In contrast to our approach, only operations that affect a single tuple are considered. Since databases are manipulated with (set-oriented) SQL commands, we consider our problem as more natural than a tuple-at-a-time approach. The detection of minimal sequences of update operations is considered in [CG97] for hierarchically structured data. The authors consider an extended set of update operations to meet the requirements of the manipulation of hierarchically structured data. The data is represented as a tree structure and there are operations that delete, copy, or move complete sub-trees. However, the corresponding update operation, i.e., to manipulate single data values, considered in [CG97] is tuple (or node)-at-a-time.

Chapter 7

Heuristics and Problem Variations

The examples in the previous chapter show that, despite pruning over 95% of the generated databases immediately, the transition graph becomes large even for tiny databases. The size of the transition graph limits applicability of the presented TRANSIT algorithms to small databases. In this chapter, we present heuristics and problem variations for finding minimal update sequences for large databases. We start by giving a classification of modification operations based on how they change the set of conflicts between a pair of databases. In variation of the original problem, we then consider only certain classes of modification operations in our algorithms for finding minimal update sequences. We prove that computing minimal update sequences is NP-hard for a certain class of operations and outline why the general problem of computing minimal update sequences appears to be NP-hard. The complexity of the problem makes heuristic approaches inevitable for large database. In the second part of this chapter, we describe two different heuristics that allow computation of update sequences for arbitrarily large databases while not necessarily finding the best (exact) solution. In our experiments, we analyze the quality of the computed results and show that even a simple greedy approach gives results of good accuracy.

7.1 A Classification of Modification Operations

The search space for minimal update sequences is enormous due to the large number of valid modification operations. For a given pair of databases the number of valid modification operations is approximately the number of modification terms times the number of closed patterns. Note that this number includes invalid modification operations that are filtered out by the algorithm described in Section 6.2.3. Our experiments in Chapter 6 show that the number of modification operations becomes very large for databases of mediocre size. In this section, we present a variation of the update distance problem. We reduce the search space by restricting the set of modification operations based on how they change the set of conflicts between a pair of databases. For example, we may only consider operations that reduce the overall number of conflicts. Our experiments show that the size of the transition graph is reduced significantly by these restrictions. We are thereby enabled to determine update sequences for larger databases than those used in the experiments of the previous chapter. The update distance for a pair of databases, however, differs depending on the class of modification operations used.

For a given a pair of databases r_1 and r_2, a modification operation $\psi_\mu = (\tau, \rho)$ alters the set of conflicts that exist between tuples in $\rho(r_1)$ and their matching partner in r_2. Let $M_\psi \subseteq M(r_1, r_2)$ denote the set of matching pairs that correspond to the tuples in $\rho(r_1)$, i.e., $M_\psi = \{m \mid m \in M(r_1, r_2) \wedge tup_1(m) \in \rho(r_1)\}$. We classify a modification operation based on how it alters conflicts between the elements of M_ψ. Let $M_{pre\psi} \subseteq M_\psi$ denote the set of matching pairs that have a conflict for attribute $attr(\tau)$ before execution of ψ_μ, i.e., $M_{pre\psi} = \{m \mid m \in M_\psi \wedge conflict(m, attr(\tau))\}$. Let $M_{post\psi} \subseteq M_\psi$ denote the matching pairs that will have a conflict for attribute $attr(\tau)$ after execution of ψ_μ, i.e., $M_{post\psi} = \{m \mid m \in M_\psi \wedge tup_2(m)[attr(\tau)] \neq value(\tau)\}$. Recall, that ψ_μ sets the value of $tup_1(m)[attr(\tau)]$ to $value(\tau)$ for each $m \in M_\psi$. In case that $tup_2(m)[attr(\tau)] \neq value(\tau)$ there will be a conflict between $\psi_\mu(r_1)$ and r_2 in the modified $tup_1(m)$ and $tup_2(m)$ after execution of ψ_μ. We divide the elements of M_ψ into four disjoint sets based on whether there is a conflict between the tuples before and after execution of ψ_μ:

- **NEUTRAL$^+$** : The set of matching pairs that do not have a conflict in attribute $attr(\tau)$ before and after execution of ψ_μ, i.e., $NEUTRAL^+ = \{m \mid m \in M_\psi \wedge m \notin M_{pre\psi} \wedge m \notin M_{post\psi}\}$.
- **NEUTRAL$^-$** : The set of matching pairs that do have a conflict in attribute $attr(\tau)$ before and after execution of ψ_μ, i.e., $NEUTRAL^- = \{m \mid m \in M_{pre\psi} \wedge m \in M_{post\psi}\}$.
- **NEW**: The set of matching pairs with newly introduced conflicts in $attr(t)$, i.e., $NEW = \{m \mid m \notin M_{pre\psi} \wedge m \in M_{post\psi}\}$.
- **SOLVED**: The set of matching pairs whose conflict in attribute $attr(\tau)$ is solved by ψ_μ, i.e., $SOLVED = \{m \mid m \in M_{pre\psi} \wedge m \notin M_{post\psi}\}$.

We call the 4-tuple of sets ($NEUTRAL^+$, $NEUTRAL^-$, NEW, $SOLVED$) the *modification fingerprint* of an operation ψ_μ for databases r_1 and r_2. Matching pairs in sets $NEUTRAL^+$ and $NEUTRAL^-$ do not represent changes to the set of conflicts other than a possible change of conflicting values for the elements in $NEUTRAL^-$. We include these sets for reason of completeness and for the definition of CLASS 3 operations below. Based on the modification fingerprint we define four different classes of modification operations:

- **CLASS 0**: The set of all valid modification operations.
- **CLASS 1**: The set of valid modification operations that decrease the overall number of conflicts, i.e., $|SOLVED| > |NEW|$. We call CLASS 1 modification operations *conflict reducer*.
- **CLASS 2**: The set of conflict reducers that decrease the overall number of conflicts and do not introduce any new conflicts, i.e., $SOLVED \neq \{\}$ and $NEW = \{\}$. We call these operations *conflict solver*.
- **CLASS 3**: The set of conflict solvers that only solve conflicts or are neutral, i.e., $SOLVED \neq \{\}$, $NEW = \{\}$, and $NEUTRAL^- = \{\}$. We call these operations *pure conflict solver*.

From our classification of modification operations it follows that CLASS 3 $\subseteq$ CLASS 2 $\subseteq$ CLASS 1 $\subseteq$ CLASS 0. We described in Section 6.2.3 how to determine the set of valid modification operations (CLASS 0). However, to determine whether an operation is of CLASS 1- CLASS 3 requires significantly more effort. We actually have to access each affected tuple and their respective matching partner. Therefore, we have to execute the operation until a modification occurs that violates the given class definition and revoke the changes. In the worst case, we have to execute the operation completely before being able to decide which class it is in.

Our hierarchical classification ensures that the number of modification operations for a given pair of databases is reduced by allowing only operations of a certain class, i.e., it holds that |CLASS 0| ≥ |CLASS 1| ≥ |CLASS 2| ≥ |CLASS 3|. To reduce the size of generated transition graphs, we change the problem definition in Section 6.1.3 to only allow operations of a certain class in minimal transformers. Figure 7-1 indicates the decrease in the number of databases tested and added to the graph when restricting the set of valid modification operations for TRANSIT-DFS on the databases used in the experiments in Section 6.3. The results show that there is already a significant drop-off in the number of databases tested and added when disabling the insertion of Skolem constants (CLASS 0 – Skolems).

TRANSIT-DFS DBP1	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0+Skolems	4,275	603,971	4,204	4,417	4,483	3
CLASS 0-Skolems	1,384	134,906	1,384	1,578	1,504	4
CLASS 1	36	104	36	38	10	4
CLASS 2	31	80	31	49	0	5
CLASS 3	31	80	31	49	0	5

TRANSIT-DFS DBP2	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0+Skolems	1,433	226,655	1,609	1,625	871	4
CLASS 0-Skolems	760	82,440	883	970	501	4
CLASS 1	105	1,012	115	102	73	4
CLASS 2	177	1,522	197	209	161	4
CLASS 3	177	1,522	197	209	161	4

TRANSIT-DFS DBP3	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0+Skolems	5,131	1,909,040	6,055	7,238	6,535	5
CLASS 0-Skolems	2,108	498,191	2,752	3,717	2,914	5
CLASS 1	458	7,960	498	631	359	5
CLASS 2	359	5,398	385	467	315	5
CLASS 3	359	5,398	385	467	315	5

TRANSIT-DFS DBP4	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0+Skolems	95	36,986	1,134	373	32	6
CLASS 0-Skolems	95	29,574	955	368	28	6
CLASS 1	112	1,377	165	259	0	6
CLASS 2	95	1,068	129	241	0	6
CLASS 3	63	696	97	147	0	6

Figure 7-1: Using the depth-first approach with different classes of modification operations.

Disallowing the insertion of Skolem constants reduces for each attribute the number of valid modification operations by approximately the number of closed patterns. The biggest reduction, however, comes from disallowing CLASS 0 operations that potentially increase the number of conflicts. Any further restriction is only marginal in our experiments. Figure 7-1 also shows that in some cases the banishment of valid modification operations may increase the effort, e.g., database pair DBP2 using CLASS 1 and CLASS 2 operations. In this case, some paths in the transition graph become invalid for operations of a more restrictive class, thereby influencing the ability to prune irrelevant databases at earlier stages. The last column in Figure 7-1 shows the update distance for each pair of databases using modification operations of the different classes.

7.2 Complexity of Computing Minimal Transformers

In the following, we prove that computing the set of minimal update sequences for a pair of databases using only CLASS 3 operations is NP-hard. We refer to the problem as TRANSIT3. In addition, we describe why the general problem of computing minimal update sequences (using CLASS 0 operations) is expected to be NP-hard as well[5].

7.2.1 Complexity using CLASS 3 Operations

Theorem 7.1: TRANSIT3 is NP-hard.

Proof: We start by showing that TRANSIT3 is in NP. We then reduce the minimum set cover problem to the problem of finding minimal update sequences for a pair of databases using only pure conflict solver. The minimum set cover problem has been shown to be NP-hard in [GJ79].

For a given pair of databases r_o and r_t, the number of different terms that are used for enumerating modification operations is finite due to the finite number of different values for each attribute (see Section 6.2.3). Recall that $A \times (r_o[A] \cup r_t[A])$ defines the set of possible terms for attribute A at any point during transformation using CLASS 3 operations. The length of modification sequences is bounded by the upper bound and we can guess a modification sequence in polynomial time. For each operation in the sequence we pick a valid subset from $\bigcup_{A \in R} (A \times (r_o[A] \cup r_t[A]))$ as modification pattern and one term as modification term. We then test whether the sequence defines a valid TRANSIT3 transformer for the given pair of databases. Therefore, TRANSIT3 is in NP. The remaining proof follows directly from Lemma 7.1-Lemma 7.4. ■

The minimum set cover problem is defined as follows: Given a universe of elements $U = \{u_1, \ldots, u_n\}$ and a set of subsets $S = \{s_1, \ldots, s_k\}$, with $s_i \subseteq U$ for $1 \le i \le k$, the goal is to find a minimum set cover $C \subseteq S$ so that every element in U is contained in $\bigcup_{s \in C} s$. We assume w.l.o.g. that no set $s \in S$ is a proper subset of any other element in S. Given an instance (U, S) of the minimum set cover problem, we define an instance of TRANSIT3 as follows: We represent (U, S) as a database r_o that contains a tuple for each element of U and an attribute for each element of S. The schema of r_o is $R(ID, A_1, \ldots, A_k, B)$ where ID is the primary key, attributes A represent the elements in S, and B is used to introduce conflicts between r_o and a database r_t described later. For each element u_i, $1 \le i \le n$, we add a tuple t_i to

[5] We like to thank *Floris Geerts* for his valuable suggestions and remarks regarding this proof.

the relation having value 1 for attribute $t_i[A_j]$ if u_i is contained in set s_j or a distinct value otherwise. The tuples in r_o contain the following values:

$$t_i[ID] = i,\ t_i[A_j] = \begin{cases} 1, if\ u_i \in s_j \\ d_i, else \end{cases}, 1 \le j \le k \text{ , and } t_i[B] = d_i, \text{ for } 1 \le i \le n.$$

We further add a tuple t_{n+1} with $t_{n+1}[ID] = n+1$ and $t_{n+1}[A_i] = t_{n+1}[B] = d_{n+1}$. Tuple t_{n+1} ensures that the empty pattern cannot be used as modification pattern later on. The *d-values* are unique for each tuple, i.e., $d_i \neq d_j$ for $1 \le i, j \le n+1$, and we request that all values are greater than 1. The uniqueness ensures that any term using these values only selects one particular tuple. We further create a database r_t that is equal to r_o except that the value in attribute B is 0 for all tuples except t_{n+1}. These are the conflicts we have to solve when transforming r_o into r_t.

Example 7-1: The example shows a pair of databases r_o and r_t resulting from a universe with 8 elements and the following set of subsets $S = \{s_1, s_2, s_3, s_4\}$, with $s_1 = \{u_1, u_2, u_3, u_4\}$, $s_2 = \{u_2, u_5, u_6\}$, $s_3 = \{u_1, u_3, u_4, u_6, u_7\}$, and $s_4 = \{u_1, u_3, u_4, u_7, u_8\}$.

r_o

ID	A_1	A_2	A_3	A_4	*B*
1	1	d_1	1	1	d_1
2	1	1	d_2	d_2	d_2
3	1	d_3	1	1	d_3
4	1	d_4	1	1	d_4
5	d_5	1	d_5	d_5	d_5
6	d_6	1	1	d_6	d_6
7	d_7	d_7	1	1	d_7
8	d_8	d_8	d_8	1	d_8
9	d_9	d_9	d_9	d_9	d_9

r_t

ID	A_1	A_2	A_3	A_4	*B*
1	1	d_1	1	1	0
2	1	1	d_2	d_2	0
3	1	d_3	1	1	0
4	1	d_4	1	1	0
5	d_5	1	d_5	d_5	0
6	d_6	1	1	d_6	0
7	d_7	d_7	1	1	0
8	d_8	d_8	d_8	1	0
9	d_9	d_9	d_9	d_9	d_9

■

We now show the correctness of this reduction. We first show that any minimal transformer for the above databases using CLASS 3 operations can be represented by a set of closed patterns (Lemma 7.1-Lemma 7.2). We then show that any such set defines a solution to the minimum set cover problem (Lemma 7.3-Lemma 7.4). Let Ψ_{CLASS3} denote any minimal transformer for the above pair of databases, i.e., $\Psi_{CLASS3}(r_o) = r_t$. The only valid modification term in Ψ_{CLASS3} is $(B, 0)$, denoted by τ_{B0}. Any other term introduces or changes conflicts violating definition of CLASS 3 operations. Thus, we may represent $\Psi_{CLASS3} = <(\rho_1, \tau_{B0}), \ldots, (\rho_l, \tau_{B0})>$, with $1 \le l \le \Delta_U(r_o, r_t)$, by the sequence $<\rho_1, \ldots, \rho_l>$ of modification patterns. Let $L_M(\Psi_{CLASS3}) = <\rho_1, \ldots, \rho_l>$ denote this sequence of modification patterns, and let $P_M(\Psi_{CLASS3}) = \{\rho_1, \ldots, \rho_l\}$ denote the corresponding set of modification patterns. Recall that $P_C(r_o)$ denotes the set of closed patterns for r_o including the empty pattern. Two patterns in $P_C(r_o)$ lead to invalid CLASS 3 modification operations: the empty pattern, and the pattern representing tuple $t\{n+1\}$. Both patterns introduce a conflict in $t\{n+1\}[B]$ when used as modification pattern. We denote the set of valid modification patterns for a database r by $P_V(r)$. The following two lemmas show that

(i) $P_M(\Psi_{CLASS3}) \subseteq P_V(r_o)$, and (ii) $P_M(\Psi_{CLASS3})$ is sufficient to represent Ψ_{CLASS3}. Let r_m denote the resulting databases after executing a sequence prefix Ψ'_{CLASS3} of Ψ_{CLASS3}.

Lemma 7.1: For any $r_m = \Psi'_{CLASS3}(r_o)$ it holds that $P_V(r_m) \subseteq P_V(r_o)$.

Proof: Assume there exists a pattern $\rho \in P_V(r_m)$ with $\rho \notin P_V(r_o)$. We distinguish between two cases:

1. If ρ contains a term τ with $attr(\tau) = B$ then ρ selects exactly one tuple in r_m. The only term for attribute B that can select more than one tuple is τ_{B0}, which would lead to an invalid modification pattern. Therefore, ρ represents one of the unchanged tuples in r_m. If this tuple is $t\{n+1\}$ then $\rho \notin P_V(r_m)$. For any other tuple there has to be the same pattern in $P_V(r_o)$ as the tuple did not change and therefore has to be in r_o as well.
2. If ρ does not contain a term τ with $attr(\tau) = B$ then ρ has to select more than one tuple in r_m. Furthermore, ρ selects the same set of tuples in r_o as it selects in r_m because all terms in ρ are satisfied by the same set of tuples in r_o and r_m. Therefore, there has to be a closed pattern $\rho' \in P_V(r_o)$ with $\rho \subseteq \rho'$. The only additional term in ρ' could to be for attribute B. In r_o, however, all values in attribute B are unique and ρ' therefore cannot contain a term for attribute B. It follows that $\rho = \rho'$ and $\rho \in P_V(r_o)$. ■

Following Lemma 7.1 the set of valid modification patterns can only shrink with any modification operation, and thus $P_M(\Psi_{CLASS3}) \subseteq P_V(r_o)$. In the following, we use P_V to denote the initial set $P_V(r_o)$.

Lemma 7.2: Any permutation of Ψ_{CLASS3} is a valid transformer for databases r_o and r_t.

Proof: The modification patterns in Ψ_{CLASS3} select all conflicting tuples to solve the conflicts between r_o and r_t. Following Lemma 7.1 modification patterns do not influence each other in that one is only available after certain modifications have been performed. Thus, any permutation of Ψ_{CLASS3} solves the same set of conflicts. Furthermore, operations cannot become obsolete by permutation. Otherwise, we could construct a minimal transformer without the modification operation containing the particular pattern. ■

Following Lemma 7.2, any permutation of $P_M(\Psi_{CLASS3})$ represents a minimal transformer for r_o and r_t. The following two lemmas show that any set $P_M(\Psi_{CLASS3})$ can be transformed into a minimal set cover $C \subseteq S$ in polynomial time.

Lemma 7.3: Any minimum set cover $C \subseteq S$ defines a minimum cover for tuples $t\{1\}, \ldots, t\{n\}$ using patterns in P_V.

Proof: For each $s_j \in S$, with $1 \leq j \leq k$, there exists a term $(A_j, 1)$ that defines a selectable subset of r_o. Therefore, there has to be a closed pattern that selects exactly the tuples corresponding to the elements in s_j. Let $P_V|s_j$ denote this pattern from P_V that corresponds to s_j. The pattern contains at least term $(A_j, 1)$. The pattern may contain additional terms if s_j contains only one element. Since no $s \in S$ is a proper subset of any other element in S, there cannot be a pattern $\rho \in P_V$ with $P_V|s_j(r_o) \subseteq \rho(r_o)$ as $\rho(r_o) = \bigcap_{(A,\,1) \in \rho} (A, 1)(r_o)$. A solution C to the minimum set cover problem defines a subset $P_U \subseteq P_V$ as $P_U = \bigcup_{s \in C} P_V|s$. The patterns in P_U completely select tuples $t\{1\}, \ldots, t\{n\}$, and there cannot be a set with fewer patterns that completely selects tuples $t\{1\}, \ldots, t\{n\}$, because all patterns in P_V only select sets of tuples corresponding to an $s \in S$, or subsets thereof. Thus, any permutation of P_U defines a minimal transformer Ψ_{CLASS3}. ■

Lemma 7.4: Any pattern set $P_M(\Psi_{CLASS3}) \subseteq P_V$ defines a minimum set cover $C \subseteq S$.

Proof: Each pattern $\rho \in P_V$ contains at least one term $(A_j, 1)$. According to the proof of Lemma 7.3, for each term $(A_j, 1)$ representing a $s_j \in S$ there exist a pattern $P_V|s_j \in P_V$ that exactly selects the tuples satisfying $(A_j, 1)$. Pattern $P_V|s_j$ selects a superset of the tuples in $\rho(r_o)$. If ρ contains more than one such $(A, 1)$-term, then there is more than one pattern selecting a superset of $\rho(r_o)$. Let $S(\rho)$ denote the subset of S defining patterns that select supersets of $\rho(r_o)$, i.e., $S(\rho) = \{s \mid s \in S \wedge P_V|s(r_o) \subseteq \rho(r_o)\}$. For each pattern $\rho \in P_M(\Psi_{CLASS3})$, we then simply select one of the elements in $S(\rho)$ to form the minimum set cover C. None of the elements in S can occur in more than one set $S(\rho)$ for the patterns in $P_M(\Psi_{CLASS3})$. Assume that for any $s \in S$ it holds that $s \in S(\rho_1)$ and $s \in S(\rho_2)$, for $\rho_1, \rho_2 \in P_M(\Psi_{CLASS3})$. In this case, we could replace ρ_1 and ρ_2 with $P_V|s$ in $P_M(\Psi_{CLASS3})$ and Ψ_{CLASS3} would not be a minimal transformer. Following Lemma 7.3, there cannot exist a set cover C' for S with $|C'| < |C|$ as we could construct a set $P_{U'}$ with $|P_{U'}| < |P_U|$ representing a set of minimal transformers having less operations than Ψ_{CLASS3}. ■

7.2.2 Complexity using CLASS 0 Operations

We now describe informally why the general problem of finding minimal transformers for a given pair of databases is expected to be NP-hard. A formal proof, however, is difficult due to the change in the set of closed patterns by modification operations that introduce conflicts and Skolem constants. For our description, we slightly modify the construction of databases r_o and r_t for a given pair (U, S). Instead of adding only a single additional tuple t_{n+1}, we add tuples $t_{n+1}, \ldots, t_{2n}$. Each of these tuples contains a single unique *d-value* for all attributes. All other tuples remain the same as before. For this pair of databases the empty pattern would introduce at least $n-1$ new conflicts and conflict groups with any modification term other than τ_{B0} without solving any of the existing conflicts. If the modification term is τ_{B0} the number of introduced conflicts and conflict groups is n. Thus, we need at least n operations to solve the new conflicts after using the empty pattern as the modification pattern. Any transformer containing the empty pattern would therefore contain at least $n+1$ operations and could not be minimal.

The problem of finding minimal update sequences remains in NP for CLASS 0 operations. The set of terms for each attribute is now $A \times (r_o[A] \cup r_t[A] \cup SKOLEM)$, with *SKOLEM* being a set of Skolem constants. The set is bounded by the length of the sequence and therefore the set of terms remains finite. The ability to introduce conflicts, however, modifies the set of closed patterns while transforming a pair of databases. In the following we argue why the introduction of conflicts is not beneficial to reduce the number of modification operations that solve the conflicts between r_o and r_t.

In general, conflicts can be introduced to increase the number of tuples selected by a term. An example was given in Figure 6-8. Increasing the set of tuples that a term selects, however, is only helpful if we are afterwards able to solve conflicts in different conflict groups using a modification pattern containing this particular term. In our setting, we could increase the tuple sets selected by terms in tuples $t\{1\}$, …, $t\{n+1\}$. Since there is only one conflict group, however, we would rather solve the conflicts immediately and avoid the additional operation to undo all newly introduced conflicts. Conflicts may also be introduced to decrease the number of tuples selected by a particular term. When using Skolem-constants, we can easily undo these conflicts with a single modification operation. This procedure is shown to be helpful in Example 6-1. By excluding tuples $t\{1\}$-$t\{3\}$ from the set of tuples selected by term $(A_3, 1)$, we can afterwards solve the conflicts in $t\{4\}$-$t\{8\}$ without introducing any further con-

flicts. The introduced conflicts in tuples $t\{1\}$-$t\{3\}$ are afterwards solved using one modification operation. Decreasing the number of tuples that satisfy a term, however, is only beneficial for terms that select tuples within a conflict group as well as tuples that do not belong to this particular conflict group. In our case the empty pattern is the only pattern that selects tuples from both subsets $\{t\{1\}, \ldots, t\{n\}\}$ and $\{t\{n+1, \ldots, t\{2n\}\}$. Thus, decreasing the matching tuples for a term is not helpful either.

We conclude that introducing conflicts is not helpful for the particular pair of databases r_o and r_t as described above. If we do not introduce conflicts, we may change existing conflicts. However, instead of changing conflicts we can solve them immediately, again saving at least one operation. Therefore, using pure conflict solvers as in TRANSIT3 appears the only solution to find minimal transformers for the described pair of databases.

7.3 Greedy TRANSIT

Heuristics are the only feasible approach to find update sequences for pairs of large databases. In the following, we present heuristics that allow computing transformers that not necessary are minimal in the number of update operations. A first simple heuristic to cope with the computational complexity of the problem is applying a greedy algorithm, called GREEDY-TRANSIT. The algorithm returns a single transformer by selecting at each level the modification operation that reduces the number of conflicts most. Given a pair of databases r_o and r_t, we start by enumerating all modification operations for r_o. For each operation ψ we determine the number of conflicts between databases $\psi(r_o)$ and r_t, i.e., the resolution distance $\Delta_R(\psi(r_o), r_t)$. The modification operation resulting in the database with the least resolution distance, i.e., the operation that reduces the number of conflicts the most, is selected as the next operation in the final transformer. Ties are broken randomly. The database resulting from the chosen operation becomes the next starting point. Again, we enumerate all modification operations and choose the operation that reduces the number of conflicts most. This is continued until the target database is reached. The described procedure ensures that the database chosen as starting point always contains fewer conflicts with r_t than any of the previous databases. Therefore, neither cycles nor duplicated databases at different levels may occur. However, the assumption that the database with the fewest conflicts has the potential of reaching the destination first is not always correct. For the databases of Figure 6-2 the resulting transformer has a length of four (shown in Figure 6-2 a)).

Figure 7-2 shows the greedy algorithm where r_s denotes the current starting point. The main challenge for the greedy algorithm is the enumeration of modification operations. Enumerating the complete set of modification operations is infeasible for large databases due to the large number of closed patterns. However, it is also not necessary. We avoid enumerating modification operations that are no candidate for the final transformer by interleaving closed pattern mining with determination of resolution distances. The algorithm is outlined in Figure 7-3. Every time a new closed pattern ρ is returned by the mining algorithm (*line 3*), we enumerate all modification operations using ρ that are able to reduce the number of conflicts more than the currently best operation ψ_{max} (*lines 4-10*). These are modification operations (i) that have a modification term that equals one of the current conflict groups $\kappa \in K(r_s, r_t)$, and (ii) select more tuples belonging to conflict group κ then the reduction in the number of conflicts by the current best operation ψ_{max} (maintained in min_{sup}). Due to the second property we can further restrict the terms and patterns in closed pattern mining. Let $sup(\tau, \kappa)$ denote the number of tuples selected by a term τ that contain a conflict belonging in conflict group κ, i.e., $sup(\tau, \kappa) = |\{ tup_I(m) \mid m \in$

$M(r_1, r_2) \wedge tup_1(m)[attr(\tau)] = tup_2(m)[attr(\tau)] \wedge tup_2(m)[attr(\kappa)] = value(\kappa)\}|$. We derive $sup(\tau, \kappa)$ while scanning the databases to determine the initial set of terms for closed pattern mining. We further define $sup(\rho, \kappa)$ as the minimum $sup(\tau, \kappa)$ for all the terms $\tau \in \rho$. This number defines the maximal number of conflicts the pattern can potentially solve from conflict group κ.

The min_{sup} value is also used as support constraint for pattern mining to avoid enumeration of patterns that do not select a sufficient set of tuples. Whenever an operation is enumerated that performs better than ψ_{max}, we are able to increase min_{sup} and thereby avoid further enumeration of patterns that cannot solve more conflicts than the new ψ_{max} (*lines 7-10*). GREEDY-TRANSIT calls *greedyNext* for each database r_s (*line 5*). The result of *greedyNext* can be empty as we use 2 as the initial min_{sup}. In this case, we solve one of the existing conflicts randomly using the tuple where the conflict occurs as the closed pattern (*line 7*).

```
1  GREEDY-TRANSIT(r_o, r_t) {
2    Ψ_T := <>;
3    r_s := r_o;
4    while(r_s ≠ r_t) {
5      ψ_next := greedyNext(r_s, r_t);
6      if (ψ_next = ⊥) {
7        ψ_next := pickRandom(r_s, r_t);
8      }
9      r_s := ψ_next(r_s);
10     append(Ψ_T, ψ_next);
11   }
12   return Ψ_T;
13 }
```

Figure 7-2: The greedy algorithm to calculate the update distance of a pair of databases. The update operation that decreases the upper bound the most calculated in subroutine *greedyNext*.

```
1  greedyNext(r_s, r_t) {
2    ψ_max := ⊥; min_sup := 2;
3    while (ρ = nextPattern(min_sup)) {
4      for each κ ∈ K(r_s, r_t) {
5        if (sup(ρ, κ) ≥ min_sup) {
6          r_c := (κ,ρ)(r);
7          if (Δ_R(r_s, r_t) − Δ_R(r_c, r_t) ≥ min_sup) {
8            ψ_max := (κ,ρ);
9            min_sup := (Δ_R(r_s, r_t) − Δ_R(r_c, r_t)) + 1;
10         }
11       }
12     }
13   }
11   return ψ_max;
12 }
```

Figure 7-3: We avoid enumerating the complete set of modification operations for large databases by interleaving pattern generation and operation enumeration.

7.4 Approximation of Update Distance

A heuristic for approximating the update distance is based on solving the conflicts within each conflict group independently. The sum of necessary operations for conflict solution of individual conflict groups is used as an approximation for the update distance. The approximated result is equal or above the lower bound, as we still need at least one modification operation per conflict group, and below or equal the upper bound, as we are still able to solve each conflict individually with a single modification operation. The presented approximation completely disregards the possible impact that the modification of values for some of the tuples may have on solving conflicts for other tuples.

Determining the minimal number of modification operations necessary to solve the conflicts within a conflict group individually still is expensive, as shown in Example 6-1. We further restrict the set of valid modification operations for approximating the update distance to CLASS 3 operations. Therefore, for solving the conflicts represented by a conflict group κ, only operations having κ as modification term are valid. The modification patterns of these operations may only select tuples from a database that are part of a conflict represented by κ or that already possess $value(\kappa)$ for attribute $attr(\kappa)$. The former is called *solution target set*, as these are the tuples that need to be modified for conflict solution, and the latter is called *solution neutral set*, as these tuples are neutral regarding the described modification operations.

Definition 7.1: Let $\kappa \in K(r_1, r_2)$ be a conflict group between a pair of databases r_1 and r_2. The *solution target set* of κ, denoted by $\xi(r_1, r_2, \kappa)$, is the set of tuples from r_1, that contain the conflicts represented by κ, i.e., $\xi(r_1, r_2, \kappa) = \{tup_1(m) \mid m \in M(r_1, r_2) \wedge tup_1(m)[attr(\kappa)] \neq tup_2(m)[attr(\kappa)] \wedge tup_2(m)[attr(\kappa)] = value(\kappa)\}$. ■

Definition 7.2: Let $\kappa \in K(r_1, r_2)$ be a conflict group between a pair of databases r_1 and r_2. The *solution neutral set* of κ, denoted by $\eta(r_1, r_2, \kappa)$, is the set of tuples from r_1 that are neutral regarding the solution of conflicts represented by κ, i.e., $\eta(r_1, r_2, \kappa) = \{t \mid t \in r_1 \wedge t[attr(\kappa)] = value(\kappa)\}$. ■

The cost for solving the conflicts represented by a conflict group κ is given by the minimal number of patterns that together select the group target set at least and the union of group target and neutral set at most. This cost forms the basis of our update distance approximation.

Definition 7.3: Given a database r and two disjoint subsets $s_t, s_n \subseteq r$. The *solution cost*, denoted by $\theta(r, s_t, s_n)$, is the minimum number of patterns $\rho_1, \ldots, \rho_q$, that select s_t completely and $s_c \cup s_n$ at most, i.e., $s_c \subseteq \rho_1(r) \cup \ldots \cup \rho_q(r) \subseteq s_c \cup s_n$. ■

Definition 7.4: The *group solution cost* for a given pair of databases r_1 and r_2, denoted by $\phi(r_1, r_2)$, is the sum of the solution cost for the conflict groups between the sources, i.e., $\phi(r_1, r_2) = \sum_{\kappa \in K(r_1, r_2)} \theta(r_1, \xi(r_1, r_2, \kappa), \eta(r_1, r_2, \kappa))$. ■

Example 7-2 shows an example. The group solution cost $\phi(r_1, r_2)$ is used as an approximation of the update distance $\Delta_U(r_1, r_2)$ of databases r_1 and r_2. Note that the approximated update distance may be above the actual update distance or below. The first case occurs, whenever there are positive side effects of solving conflicts in one attribute for solving conflicts in other attributes. The latter occurs, whenever the respective modification operations interfere with each other, i.e., after executing one of them, the other is no longer executable or has a different result. The group solution cost may also be

used as a replacement for the lower bound within the algorithms TRANSIT-BFS and TRANSIT-DFS. Using an approximated lower bound in a branch and bound approach may imply that the exact solution is missed. However, in all our experiments this heuristic computed the exact solution. The according algorithms are called TRANSIT-BFS (GS) and TRANSIT-DFS (GS), respectively. We can also use the group solution cost in a greedy approach. Instead of choosing the operation reducing the number of conflicts the most, we choose the operation that reduces the group solution cost the most. We thereby enable the usage of Skolem-constants that are otherwise omitted. However, in order to determine the next starting point in the greedy algorithm, we now need to enumerate the complete set of modification operations. The corresponding algorithm is called GREEDY-TRANSIT (GS) in our experiments.

Example 7-2: The group solution cost for the given pair of databases having four conflict groups is 8. For each conflict group we list the group solution cost and respective patterns. For example, conflict group κ_2 contains tuples $t\{3\}$, …, $t\{6\}$. Pattern $\rho_{2,1}$ selects tuples $t\{3\}$ and $t\{4\}$, pattern $\rho_{2,2}$ selects tuple $t\{5\}$, and pattern $\rho_{2,3}$ selects tuple $t\{6\}$.

r1

A_1	A_2	A_3	A_4	A_5	A_6
1	1	2	3	1	1
2	1	3	3	1	0
3	1	2	1	0	0
4	1	2	2	1	0
5	1	2	7	1	1
6	1	2	6	1	1
7	2	2	5	1	1
8	0	2	6	1	1

r2

A_1	A_2	A_3	A_4	A_5	A_6
1	2	2	6	1	1
2	2	3	6	1	0
3	2	3	1	0	0
4	2	3	2	0	0
5	2	3	7	0	1
6	2	3	6	0	1
7	2	2	5	1	1
8	0	2	6	1	1

$\kappa_1 = (A_2, 2)$: $\theta(r_1, \xi(r_1, r_2, \kappa_1), \eta(r_1, r_2, \kappa_1)) = 1$ with $\rho_{1,1} = \{(A_2, 1)\}$.

$\kappa_2 = (A_3, 3)$: $\theta(r_1, \xi(r_1, r_2, \kappa_2), \eta(r_1, r_2, \kappa_2)) = 3$ with $\rho_{2,1} = \{(A_2, 1), (A_3, 2), (A_6, 0)\}$, $\rho_{2,.2} = terms(t\{5\})$, $\rho_{2,3} = terms(t\{6\})$.

$\kappa_3 = (A_4, 6)$: $\theta(r_1, \xi(r_1, r_2, \kappa_3), \eta(r_1, r_2, \kappa_3)) = 1$ with $\rho_{3,1} = \{(A_4, 3)\}$.

$\kappa_4 = (A_5, 0)$: $\theta(r_1, \xi(r_1, r_2, \kappa_4), \eta(r_1, r_2, \kappa_4)) = 3$ with $\rho_{4,1} = terms(t\{4\})$, $\rho_{4,3} = t\{5\}$, $\rho_{4,3} = terms(t\{6\})$.

■

Computing the exact solution cost is similar to the problem described in Section 7.2 and therefore NP-hard. We therefore implemented a greedy approach shown in Figure 7-4. The calculation starts by determining the set of modification patterns that (i) only select tuples from $\xi(r_1, r_2, \kappa) \cup \eta(r_1, r_2, \kappa)$, and (ii) select at least one tuple from $\xi(r_1, r_2, \kappa)$ (*lines 3-8*). The empty pattern is denoted by ρ_e, and $\xi(r_1, r_2, \kappa)$ and $\eta(r_1, r_2, \kappa)$ by s_t, s_n, respectively. We then choose greedily the pattern ρ_{max} that selects the largest subset from s_t (*line 10*). We remove ρ_{max} from P_{valid} and $\rho_{max}(s_t)$ from s_t. The algorithm terminates when s_t is empty. The algorithm for computing the group solution cost for a pair of databases, called TRANSIT-APPROX, calls GREEDY-SOLUTION-COST for each conflict group and summates the results.

```
1 GREEDY-SOLUTION-COST(r, s_t, s_n) {
2   scost := 0;
3   P_valid := P_C(r) ∪ {ρ_e};
4   for each ρ ∈ P_valid do {
5     if ((ρ(r) ⊂ s_n) || (ρ(r)/ (s_t ∪ s_n) ≠ ∅)) {
6       P_valid := P_valid / {ρ};
7     }
8   }
9   while (s_t ≠ ∅) {
10    ρ_max := max_select(P_valid, s_t);
11    s_t := s_t / ρ_max(s_t)
12    P_valid := P_valid / {ρ_max};
13    scost++;
14  }
15  return scost;
16}
```

Figure 7-4: A greedy algorithm for calculating the solution cost.

7.5 Experimental Results

We now discuss the results of our experiments for the described heuristic approaches. We again use the database pairs that were used for our experiments in Section 6.3. We start by comparing the effort and accuracy of the greedy approaches and the approximation approach with the exact algorithms.

7.5.1 Accuracy of the Heuristic Approaches

TRANSIT-DFS (GS) and TRANSIT-BFS (GS)

Figure 7-5 shows the necessary effort to determine the set of minimal transformers using the group solution cost as the lower bound. In our experiments, this heuristic always computes the correct update distance. The resulting transition graphs are in general smaller in size than those found by TRANSIT-BFS and TRANSIT-DFS in Chapter 6 (the exception is DBP4). The missing vertices and edges within the final transition graph result in missing some of the minimal transformers. The total number of minimal transformers in the resulting transition graphs is denoted by |T| and shown in the last column of Figure 7-5. Compared to the numbers in Figure 6-10 a), the effort for the heuristic approach is less than for the according exact approach. The improvement is especially significant for the first three database pairs where the number of databases tested and generated is reduced by up to 99%. The improvement is only marginal for the database pair DBP4 where the lower bound actually equals the update distance while our approximation has a larger value. As a downside, the computation cost may increase due to the computation of the group solution cost. This is especially true for TRANSIT-BFS (GS), where the number of databases tested is larger than for TRANSIT-DFS (GS). Figure 7-6 compares the execution time of the two exact approaches with TRANSIT-DFS (GS). Despite the extremely high accuracy, the computation cost (and not the memory requirements) prevents us from applying this heuristic to larger databases.

GREEDY-TRANSIT and TRANSIT-APPROX

Figure 7-7 lists the resulting update distances for the four database pairs using the greedy approaches with different scoring functions compared to the approximation of the update distance. GREEDY-TRANSIT (GS) determines the optimal update distance for each of the databases. However, the approach is limited to smaller and medium sized databases due to high computational cost and the necessity to enumerate the complete set of valid modification operations.

TRANSIT-BFS (GS)	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U	|T|
DBP1	4	499	27	2	0	3	1
DBP2	22	3,685	172	31	12	4	24
DBP3	957	391,067	7,319	6,092	3,042	5	72
DBP4	5,049	2,232,558	31,956	54,843	22,657	6	1,500

TRANSIT-DFS (GS)	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U	|T|
DBP1	3	402	23	2	0	3	1
DBP2	16	2,648	124	25	0	4	24
DBP3	18	7,226	803	221	34	5	72
DBP4	83	32,221	1,009	329	26	6	1,500

Figure 7-5: The necessary effort with group selection cost as lower bound. The last two columns show the resulting update distance and the number of transformers in the resulting transition graph.

	TRANSIT-BFS	TRANSIT-DFS	TRANSIT-DFS (GS)
DBP1	366	4,311	89
DBP2	399	1,490	645
DBP3	236,596	12,747	3,621
DBP4	40,329	282	16,836

Figure 7-6: Execution time (in *ms*) for the exact algorithms and TRANSIT-DFS (GS).

	GREEDY-TRANSIT (UB)	GREEDY-TRANSIT (GS)	TRANSIT- APPROX
DBP1	4	3	5
DBP2	5	4	6
DBP3	7	5	8
DBP4	7	6	6

Figure 7-7: The resulting update distance computed by greedy approaches using the database with the lowest upper bound and lowest group solution cost as the next starting point. The results are compared to the approximation of the update distance using TRANSIT-APPROX.

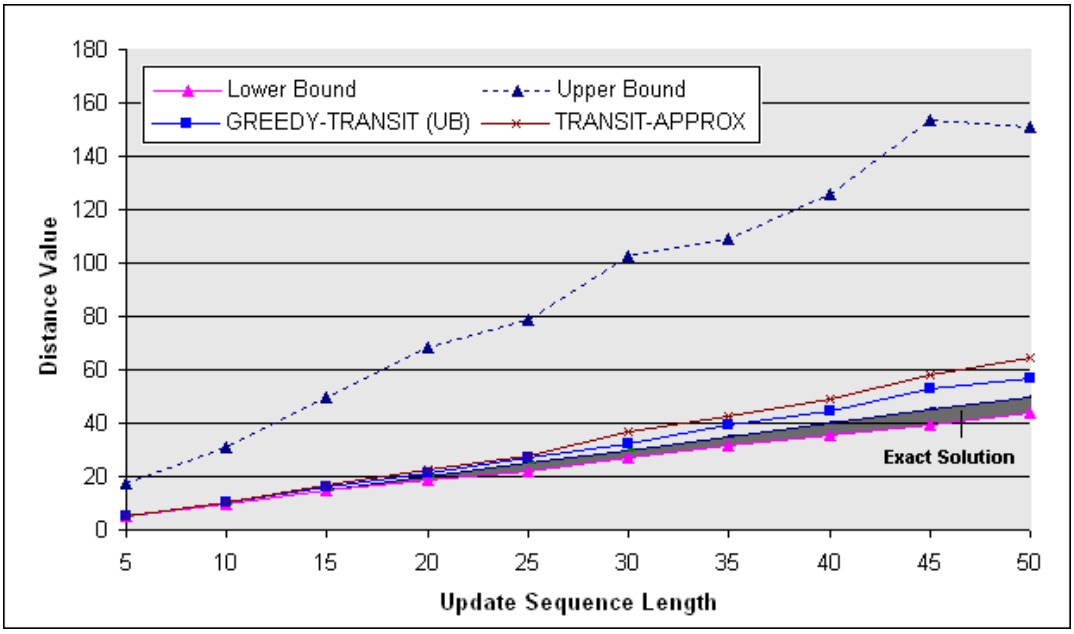


Figure 7-8: Comparing the accuracy of GREEDY-TRANSIT (UB) and TRANSIT-APPROX for various pairs of contradicting databases generated using update sequences containing between 5 and 50 operations.

In order to assess the accuracy of the greedy algorithm and of the update distance approximation we use a database of 10 attributes and 100 tuples and modify it using arbitrary update sequences of length between 5 and 50. We then compute the update distance between the original and the resulting database using the two algorithms GREEDY-TRANSIT (UB) and TRANSIT-APPROX. The results are shown in Figure 7-8. All values are averaged over ten runs. The dark grey area above the lower bound (labeled *Exact Solution*) highlights the location of the exact solution that has to be between the lower bound and the length of the sequences that generated the contradicting databases. Note that the sequence that generated the database not necessarily has to be minimal and the complexity of the problem prevents us from computing the exact solution. The greedy approach and the approximation are both surprisingly accurate for short update sequences. For longer update sequences the accuracy decreases but remains in reasonable bounds. Overall, the greedy approach outperforms the approximation in accuracy. On the other hand, the execution time for TRANSIT-APPROX is only a few milliseconds for the tested database while for the GREEDY-TRANSIT (UB) it is between 875 - 74,000 *ms* on a CITRIX METAFRAME™ Server containing two Intel Xenon 2,4 GHz processors and 4 GB main memory.

When generating the contradicting databases for the accuracy experiments, we randomly chose one operation from the set of valid modification operations for the current database. The accuracy of GREEDY-TRANSIT (UB) and TRANSIT-APPROX decreases if we restrict the chosen modification operation to affect a minimum of n tuples. Figure 7-9 shows the update distances computed by GREEDY-TRANSIT (UB) when allowing only modification operations whose patterns select at least 2, 5, 10, or 20 tuples. Also shown is the resulting upper bound, i.e., number of conflicts, for the generated databases. Using patterns with higher selectivity increases the number of conflicts between the resulting databases without increasing the length of the generating sequences. While the accuracy decreases, the results are still closer to the actual update distance then the upper bound.

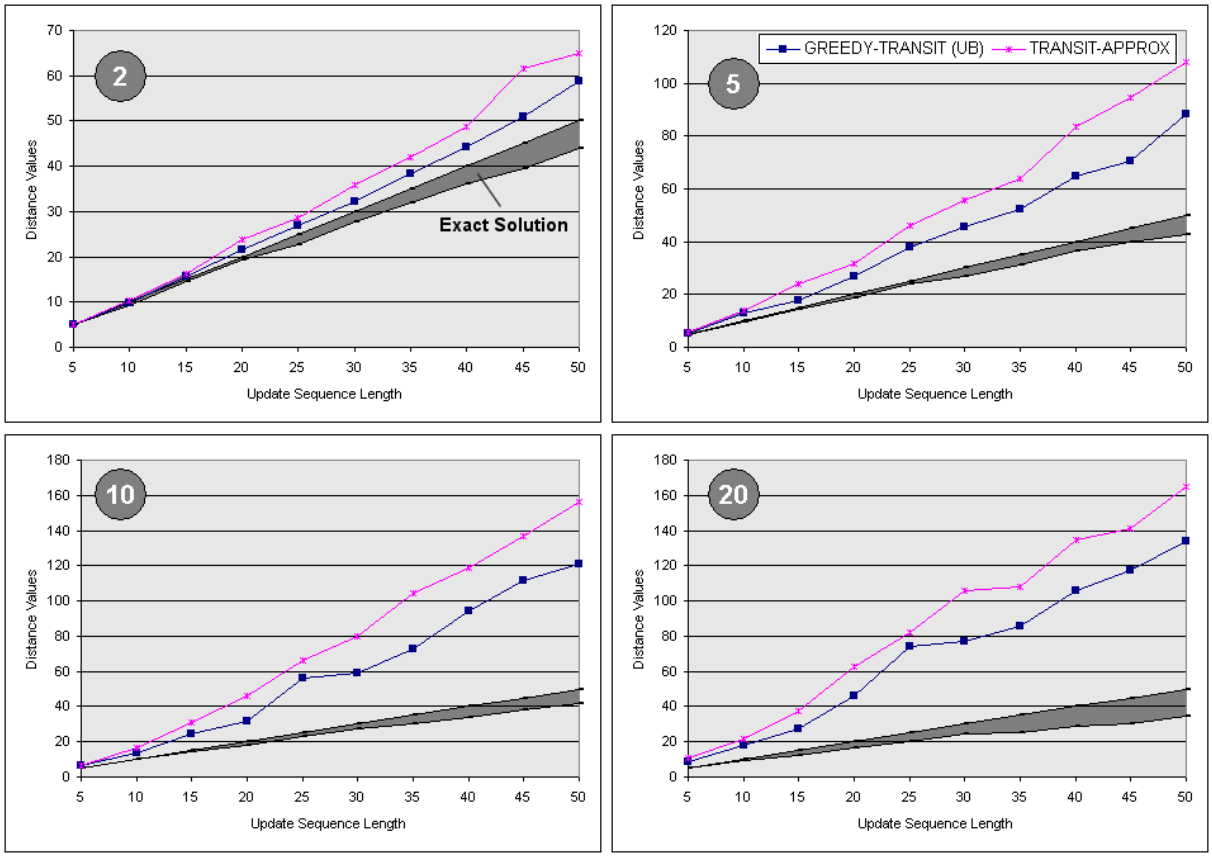


Figure 7-9: The accuracy of GREEDY-TRANSIT (UB) compared to TRANSIT-APPROX for sequences with operations having varying pattern selectivity of 2, 5, 10, and 20.

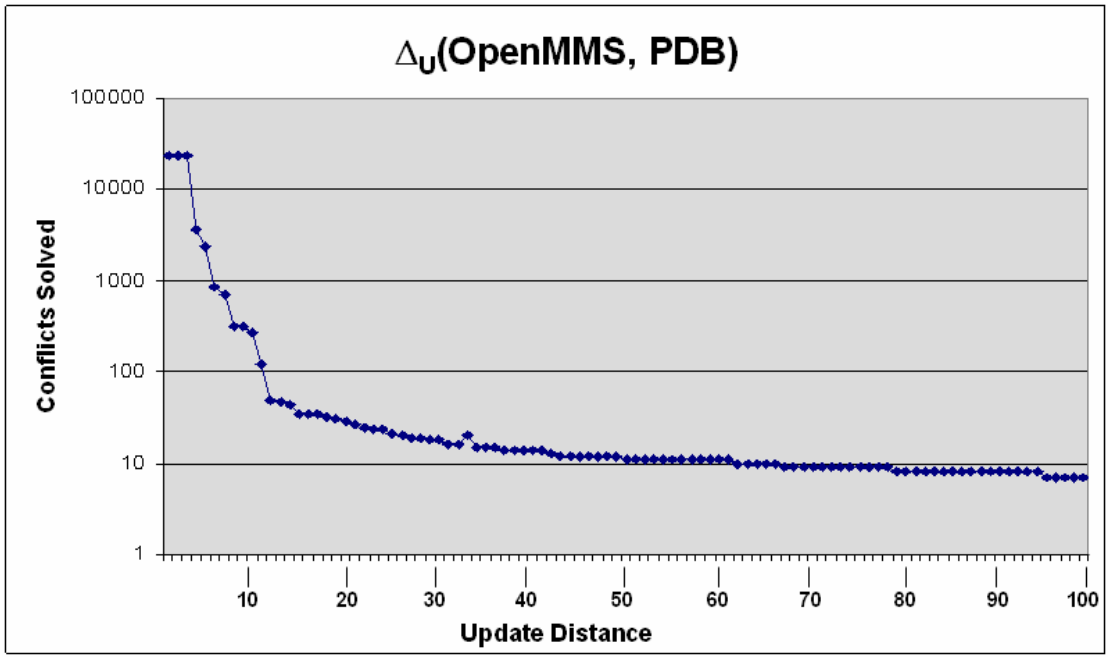


Figure 7-10: The figure shows the number of conflicts solved by the first 100 operations in the update sequence for databases OPENMMS and PDB. The total sequence contains 15,267 operations.

We applied GREEDY-TRANSIT (UB) on the protein structure databases OPENMMS and PDB having over 20,000 tuples each and nearly 100,000 conflicts between them. The resulting update sequence contained 15,267 update operations and computation took more than 24 hours. The result in Figure 7-10 shows that over 97% of the operations in the sequences solved less than 10 conflicts. This effect might be an argument for the disadvantage of the greedy approach. However, it more likely points towards the fact that a large number of arbitrary conflicts exists between the databases that do not follow a systematic reason. By interpreting the operations at the start of the sequence, we discovered update operations that describe the commonly known systematic differences between the databases, like usage of different value representations or vocabularies in some of the attributes.

7.6 Summary

Within this chapter we discussed why computing minimal transformers is both, computationally expensive and memory consuming. The complexity makes application of algorithms presented in Chapter 6 impossible for almost any pair of real-world databases. We describe a problem variation that reduces the number of valid modification operations for each database and thereby the size of the transition graph. For large databases, however, constructing the transition graph also becomes infeasible for these variations. We prove that for a very restrictive class of modification operations computation of minimal transformers for a pair of databases is NP-hard already.

The greedy heuristics GREEDY-TRANSIT(UB) is currently the only approach able to compute update sequences for large databases. Our experiments show that the accuracy of this greedy approach is surprisingly good. In general, the problem of computing minimal transformers is closely related to the minimum set cover problem. Additional complexity arises from the fact that the set of selectable tuples in form of closed patterns changes with every modification operation. The greedy algorithm has been shown to have good accuracy for the minimum set cover problem in [Sla96]. These results further strengthen our belief that a greedy algorithm is valuable approach for our problem. However, we are currently lacking any theoretical assertions or bounds on the accuracy of the greedy approach. Finding such bounds is considered future work.

Approximation of update distance using group solution cost yields good experimental results when used as the weight function in a greedy approach or as the lower bound in any of the branch and bound approaches. However, our current algorithm for computing this approximation relies on computing the complete set of closed patterns first, thus limiting the algorithms to small databases. The computational cost in the greedy approach could be reduced by computing closed patterns for each conflict group independently. Thereby, we could limit the terms to those that select tuples in that particular conflict group. However, for large conflict groups we would still have to enumerate a large number of patterns. The computational cost for TRANSIT-BFS(GS) and TRANSIT-DFS(GS) could be reduced significantly if we were able to compute the approximation for any derived database based on the set of closed patterns for the original database and the modification operation used to derive a database. A solution to this problem is also considered future work.

Chapter 8

Conclusion

In this chapter we conclude by giving a summary of our achievements for identifying and describing systematic differences in overlapping databases. We then give a brief outlook on further applications for minimal update sequences.

8.1 Summary

Data of high quality is of uttermost importance in scientific application. Especially poor data accuracy can have devastating consequences on the success of research projects and for those who rely on the outcome. In this thesis we argue that existing data cleansing efforts, primarily developed for business domains, insufficiently address the problem of semantic data cleansing in general, and ensuring high accuracy for scientific data in particular. We give convincing arguments that data merging based on identification of systematic conflict classes is both useful and feasible for semantic data cleansing in scientific databases.

Data Merging for Semantic Data Cleansing

There exist several applications and tools that deal with data cleansing and the problem of improving the quality of existing databases. In Chapter 2, we give a classification of existing data cleansing methods. We first define a set of data deficiencies for a given database that diminish data quality. From existing definitions of data quality we then extract a set of criteria that are affected by the defined data deficiencies. We thereby take a data centric approach that focuses on the quality of the values and the database instance and ignores subjective quality criteria like believability and reputation. We discuss the most common methods used in data cleansing and specify for each method the quality criteria affected by it. Our classification of data deficiencies, data quality criteria, and data cleansing methods allows for better comparison and evaluation of existing and future data cleansing approaches. A comparison of existing data cleansing methods and approaches reveals that (a) the problem of semantic data cleansing is only poorly addressed by existing approaches, and (b) the combination of duplicate detection and conflict resolution has great potential for semantic data cleansing. Our review of existing data cleansing projects shows that support for conflict resolution is not considered in any of these projects. We therefore argue that the methods presented in this thesis pose valuable extensions to existing data cleansing frameworks.

Data Quality in Genome Databases

Practical experience and quality studies show that scientific databases often do not meet the standards of high data quality. Existing data cleansing approaches, developed primarily for business applications, are mainly concerned with producing a unified and consistent data set, addressing primarily syntactical problems and ignoring the semantic problem of verifying the correctness of the represented information. In Chapter 3, we show that errors in genome data are caused by inadequacies of the data production process. We give a description of the general production process for genome data. Through careful analysis of the experimental and annotation process of genome data, we identify five classes for poor data quality. We identify the producers of these errors and pinpoint the employment of each of these producers in the data production pipeline and the types of error they produce. Our analysis provides a sound basis for quality improvement efforts. We discuss the ability for data cleansing in each step of the data production process and list available cleansing approaches. Only prohibitively expensive quality checking within the process, using quality checking modules, can increase quality during data production. Existing approaches are primarily focused on ensuring high quality for sequence data. Thus, the problem of how to eliminate existing errors in annotation data after data production remains to be solved.

Data Cleaning in Scientific Databases

Our discussion of quality checks and cleansing abilities for genome data shows that mainly statistical methods are used for genome data cleansing. In Chapter 3, we outline our experimental studies on semantic data cleansing for genome data based on integrity constraint repair and data merging. In our first study, we use a well known constraint on translation of DNA sequences to identify inaccurate annotations in a genome database. We develop a re-annotation method to replace the inaccurate annotations with more accurate ones. In our second study, we utilize overlapping databases on protein structure annotations for data cleansing. We identify systematic differences between these databases and utilize the information for conflict resolution. Both studies show the ability to enhance the accuracy of existing genome databases. However, the first approach not only requires the necessary domain knowledge to specify an appropriate re-annotation strategy but also significant effort for software development. In contrast, the second approach can rely on existing data integration techniques and does not require a domain expert to have additional computer science skills.

Data Merging and Context-aware Conflict Resolution

Data merging relies on our ability to identify conflicts and solve them effectively. The second problem has received little research attention so far. Context-aware conflict resolution is predominant approach to solve groups of conflicts that follow the same systematic reason; a frequent scenario in scientific data sources. Context-aware conflict resolution requires identification of systematic conflicts between given databases. In Chapter 4, we describe algorithms for comparing pairs of overlapping databases that find interesting conflicts that occur systematically or follow certain patterns. Based on the popular concept of association rules, we define contradiction patterns that provide a quick way to find characteristic data properties occurring in conjunction with value conflicts between databases. We define interestingness measures that allow restriction of pattern properties we are interested in. Conflicts are classified based on contradiction patterns and the patterns act as descriptive information providing insights towards potential conflict reasons. Both, conflict classes and their descriptive information are valuable in support of context-aware conflict resolution.

Retrospective Documentation of Conflict Generation

Contradiction patterns are based on a simple conflict model. In Chapter 5, we present a process-oriented conflict model, named the ancestor-based conflict model. The ancestor-based conflict model regards a pair of overlapping databases as modified copies of a common ancestor database. Modification of databases is performed using sequences of set-oriented update operations. We argue that the ancestor-based conflict model is very well capable of describing systematic differences between databases that either result from systematic modification using SQL-like update operations or result from systematic differences in the data production process. We present an algorithm that is capable of identifying, in retrospective, modification operations in the potential modification sequences of a given pair of databases. These operations are represented by condition-action pairs and present a natural approach of describing systematic conflicts. We give a classification of modification operations based on the action they define and allow to restrict pattern mining to operations of certain classes. Focusing on certain operation classes is desirable since the conflicts resulting from different classes often require specific individual conflict resolution functions.

Minimal Update Sequences

Following the ancestor-based conflict model, we define sequences of set-oriented modification operations as comprehensive descriptions for systematic conflicts between overlapping databases in Chapter 6. Compared to contradiction patterns update sequences define groups of conflicts over the complete set of contradictions and not only for those in one attribute. Update sequences are also able to outline dependencies that exist between conflicts in different attributes that are not revealed when mining patterns for individual attributes only. We call the number of operations in minimal update sequences the update distance for databases. We consider the update distance a semantic distance measure, as it is inherently process-oriented in contrast to purely syntactic measures such as counting differences. We derive upper and lower bounds for the update distance of databases and present branch and bound algorithms for calculating minimal update sequences for a pair of databases. However, finding the exact solution for the update distance is only feasible for very small databases. In Chapter 7, we prove that the computational complexity for determining the update distance using a restricted set of operations is NP-hard. To cope with the complexity of the problem, we present different heuristics and an approximation of the distance that allow computation of update distances for almost arbitrary databases at surprisingly good accuracy.

Overall, the algorithms presented in the second and third part of this thesis provide a sound basis for context-aware conflict resolution. Especially for scientific data, overlapping data sources often provide the main source of information for data cleansing by data merging. Contradiction patterns, conflict generators, and update sequences provide different representations of conflicts that share characteristic data properties and thereby point towards a systematic conflict background. Each representation is regarded a summarization or description of systematic conflict properties. Evaluated by an expert user these conflict descriptions allow to classify existing conflicts, assess the quality of different sets of conflicting values, and define an effective context-aware conflict resolution strategy. Furthermore, the given information point towards possible pitfalls in the data production and processing workflow and can be used for process improvement to avoid future inconsistencies.

8.2 Outlook

We motivated our definition of the update distance for overlapping databases by an analogous definition of the edit distance in sequence analysis. The update distance is considered a semantic, process-oriented distance measure. Minimal update sequences give valuable clues on what has happened to make a pair of database different from its original state. Based upon the update distance, we define two additional distance measures for pairs of contradicting databases. These definitions are motivated by the following two questions:

- *How did a pair of databases evolve from a common ancestor?* This question results directly from the ancestor-based conflict model. However, minimal update sequences between the given databases only represent approximations of the original operations.
- *How can we transform a pair of databases into a common descendant?* This question is related to the problem of integrating a pair of databases.

The databases and modification processes surrounding these questions are depicted in Figure 8-1. The first question follows the assumption of the ancestor-based conflict model that a given pair of databases r_1 and r_2 evolved as modified copies of a common ancestor r_a. The modifications where performed by applying sequences of update operations Ψ_{L_1} and Ψ_{L_2} to copies of the ancestor r_a. This approach is related to the phylogeny of organisms, i.e., the evolution from a common ancestor by evolutionary events like the modification of the DNA sequence. Similar to this evolutionary process, we describe the *process of divergence* of r_1 and r_2 from r_a by the triple $(r_a, \Psi_{L_1}, \Psi_{L_2})$, with $\Psi_{L_1}(r_a) = r_1$ and $\Psi_{L_2}(r_a) = r_2$.

In [CWO+04] the phylogenetic distance between two organisms is defined as the total number of intermediate organisms along the lines of descent leading to their most recent common ancestor. For overlapping databases, the phylogenetic distance describes the minimal number of intermediate states for their divergence from a common, but probably unknown, ancestor. Based on Lemma 6.1 any database r from $\mathfrak{R}(R)$, i.e., the infinite set of databases following schema R that satisfy the primary key constraint, is a common ancestor for a pair of databases, as there exists at least one transformer that transforms r into any other database from $\mathfrak{R}(R)$. We assume the simplest, i.e., shortest transformer, to be the most likely explanations of the observed differences.

Definition 8.1: For a pair of databases r_1 and r_2, the *phylogenetic distance*, denoted by $\Delta_P(r_1, r_2)$, is defined as the minimal number of update operations necessary to derive r_1 and r_2 from any of the possible ancestors by independent application of a pair of update sequences, i.e., $\Delta_P(r_1, r_2) = \forall\, r_a \in \mathfrak{R}(R) : min(\Delta_U(r_a, r_1) + \Delta_U(r_a, r_2))$. ■

The challenge with determining the phylogenetic distance is to find those databases from $\mathfrak{R}(R)$, for which the sum of the update distances is minimal. An application for the phylogenetic distance is for example in database forensics. Imagine a scenario where we are given a set of databases that origin from the same ancestor by copying and modifying the original data. The goal is to identify who copied from whom in this scenario. By constructing a phylogenetic tree of the given databases based on the phylogenetic distance, we are able to provide the most probable answer to this question.

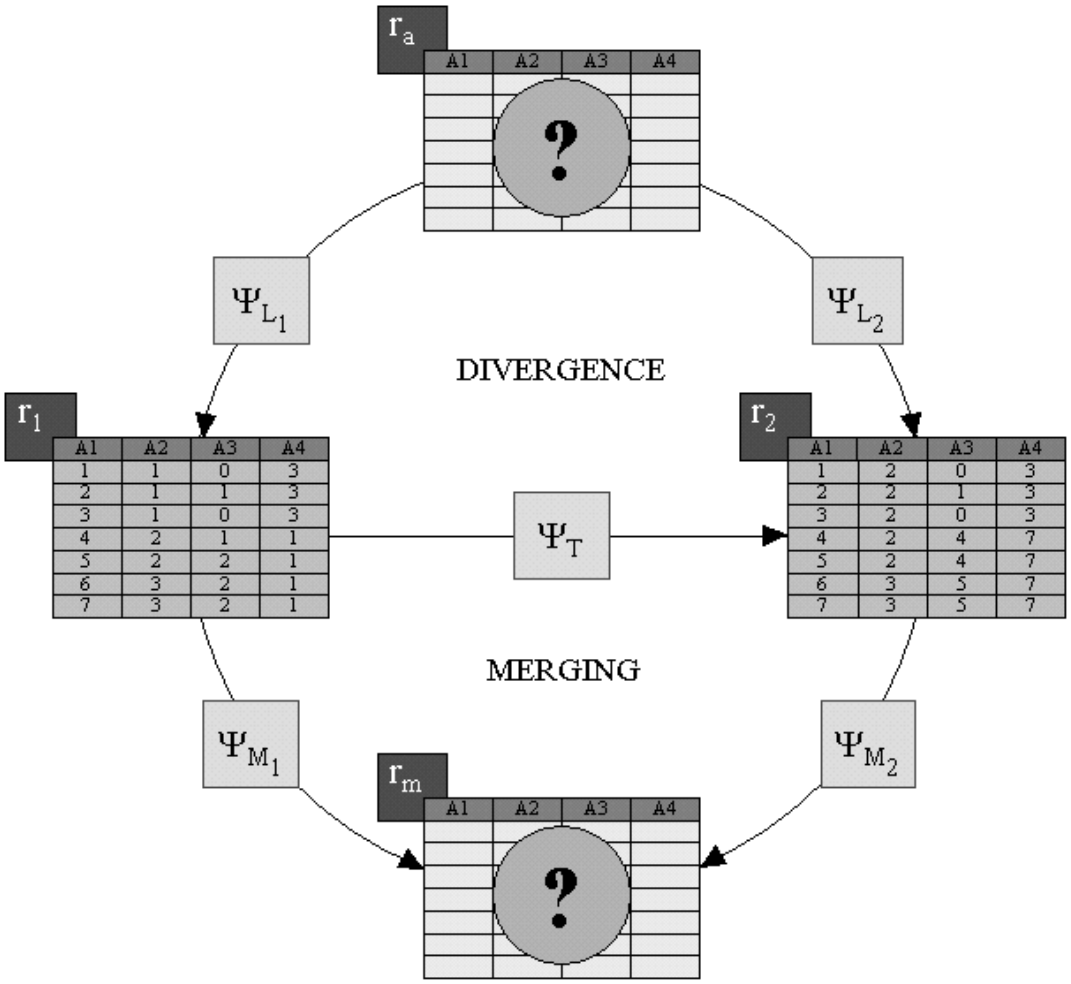


Figure 8-1: Distance measures in the evolution of databases.

The second of the above question results from the problem of data integration. When integrating or merging two databases, we need to solve the conflicts between them. We thereby assume a proceeding where we derive an integrated database by retaining existing values from each of the original databases. Therefore, the resulting database contains within each tuple and each attribute one of the possibly two values for this attribute from the matching partners. Tuples without a matching partner are added to the merged database as they are.

Definition 8.2: For a pair of databases r_1 and r_2, a *merged database* r_m is defined as (i) the union of the tuples without a matching partner from either source and (ii) the overlapping part of r_1 and r_2 with conflicts solved by a set of resolution function F that chose one of the conflicting values, i.e., $r_m = U(r_1, r_2) \cup U(r_2, r_1) \cup F(C(r_1, r_2))$. ■

In general, a resolution function $f \in F$ takes two or more values from a certain domain and returns a single value of the same domain. Any of these resolution functions completely solves the conflicts within an attribute when applied on the whole database. The resolution functions that we consider here are modification operations as defined above, i.e., they are only applied to those tuples that satisfy a given condition in form of a pattern. Therefore, it holds that:

- Each tuple contained in one of the databases r_1 and r_2 is also contained in r_m.
- Attribute values for tuples in r_m are derived from values of the corresponding tuples in r_1 or r_2.

We describe the transformation of each of the databases r_1 and r_2 into r_m by update sequences. The process of *merging a pair of databases* r_1 and r_2 into r_m is defined by the triple $(r_m, \Psi_{M_1}, \Psi_{M_2})$, where r_m is a common descendant of r_1 and r_2 and Ψ_{M_1} and Ψ_{M_2} describe the transformation of r_1, respectively r_2, into r_m, i.e., $\Psi_{M_1}(r_1) = r_m$ and $\Psi_{M_2}(r_2) = r_m$. Several databases from $\mathfrak{R}(R)$ fulfill the described constraints of a merged database for a pair of databases. We again regard the merged database resulting from the shortest sequences of update operations as the most likely one.

Definition 8.3: The *integration or merge distance* of a pair of data sources r_1 and r_2, denoted by $\Delta_M(r_1, r_2)$, is defined as the minimal number of update operations necessary in order to transform the sources into a merged database. Let $\mathfrak{R}_{valid}(r_1, r_2)$ denote the set of databases fulfilling the constraints of a merged database of r_1 and r_2. The integration distance is then defined as $\Delta_M(r_1, r_2) = \forall\ r_m \in \mathfrak{R}_{valid}(r_1, r_2)$: $min(\Delta_U(r_1, r_m) + \Delta_U(r_2, r_m))$. ■

Update sequences resulting from the merge distance provide valuable information regarding possible conflict resolution strategies for a given pair of databases. Similar to the definition of the integration distance, we are also interested in finding a context-aware resolution strategy that is minimal in the number of operations executed. Both distance measure presented in this outlook, however, rely on our ability to efficiently determine the update distance between multiple databases, a task that has been shown infeasible for pairs of databases exceeding very small number of attributes and tuples. Definition of appropriate algorithms for these advanced update distances therefore remains an open problem.

Bibliography

[ABC99] M. Arenas, L. Bertossi, J. Chomicki. Consistent Query Answers in Inconsistent Databases. Proc. ACM Symposium on Principles of Database Systems (PODS), Philadelphia, Pennsylvania, 1999.

[ACG02] R. Ananthakrishna, S. Chaudhuri, V. Ganti. Eliminating Fuzzy Duplicates in Data Warehouses. Proc. Int. Conf. On Very Large Data Bases (VLDB), Hong Kong, China, 2002.

[ACM+99] S. Abiteboul, S. Cluet, T. Milo, P. Mogilevsky, J. Siméon, S. Zohar. Tools for Data Translation and Integration. Bulletin of the Technical Committee on Data Engineering, Vol. 22, *No. 1*, 1999, 3-8.

[ACT05] A. Rodríguez, J.M. Carazo, O. Trelles. Mining association rules from biological databases. Journal of the American Society for Information Science and Technology, Vol. 56, *Issue 5*, 2005, 493-504.

[AFV94] M.D. Adams, C. Fields, J.C. Venter. Automated DNA Sequencing and Analysis. Academic Press, ISBN 0127170103 2004, 1994.

[AIS93] R. Agrawal, T. Imieliński, A. Swami. Mining association rules between sets of items in large databases. Proc. ACM International Conference on Management of Data (SIGMOD), Washington, D.C., United States, 1993, 207-216.

[AS94] R. Agrawal, R. Srikant. Fast Algorithms for Mining Association Rules. Proc. Int. Conf. On Very Large Data Bases (VLDB), Santiago de Chile, Chile, 1994.

[AS95] R. Agrawal, R. Srikant. Mining sequential patterns. Proc. Int. Conference on Data Engineering (ICDE), Taipei, Taiwan, 1995.

[AU79] A.V. Aho, J.D. Ullman. Principles of Compiler Design. Addison-Wesley Publishing Company, ISBN 0-201-00022-9, 1979.

[Bay98] J. Bayardo, Jr. Efficiently mining long patterns from databases. Proc. ACM International Conference on Management of Data (SIGMOD), Seattle, Washington, United States, 1998, 85-89.

[BB95] S. Bell, P. Brockhausen. Discovery of Data Dependencies in Relational Databases. Proc- of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases, 1995, 53-58.

[BBA+03] B. Boeckmann, A. Bairoch, R. Apweiler, M. Blatter, et al. The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. Nucleic Acids Research, Vol. 31, *No. 1*, 2003, 365-370.

[BBB+05] A. Bilke, J. Bleiholder, C. Böhm, K. Draba, F. Naumann, M. Weis. Automatic Data Fusion with HumMer. Proc. Int. Conf. On Very Large Data Bases (VLDB), Trondheim, Norway, 2005.

[BBF+01] T.N. Bhat, P. Bourne, Z. Feng, G. Gilliland, S. Jain, V. Ravichandran, B. Schneider, K. Schneider, N. Thanki, H. Weissig, J. Westbrook, H.M. Berman. The PDB data uniformity project. Nucleic Acid Research, Vol. 29, *No. 1*, 2001, 214-218.

[BCG01] D. Burdick, M. Calimlim, J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. Proc. of the Int. Conf. on Data Engineering (ICDE), Heidelberg, Germany, 2001.

[BDF+03] H. Boutselakis, et al. E-MSD: the European Bioinformatics Institute Macromolecular Structure Database. Nucleic Acid Research, Vol. 31, *No. 1*, 2003, 458-462.

[BFFR05] P. Bohannon, W. Fan, M. Flaster, R. Rastogi. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modifications. Proc. ACM International Conference on Management of Data (SIGMOD), Baltimore, Maryland, United States, 2005.

[BGK+06] O. Benjelloun, H. Garcia-Molina, H. Kawai, T.E. Larson, D. Menestrina, Q. Su, S. Thavisomboon, J. Widom. Generic Entity Resolution in the SERF Project. IEEE Data Engineering Bulletin, 2006.

[BKM+07] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, D.L. Wheeler. GenBank. Nucleic Acids Research, Vol. 35, 2007, D21-D25, doi:10.1093/nar/gkl986.

[BKW+77] F.C. Bernstein, T.F. Koetzle, G..J.B. Williams, E.F. Meyer Jr., M.D. Brice, J.R. Rodgers, O. Kennard, T. Shimanouchi, M. Tasumi. The Protein Data Bank: a computer-based archival file for macromolecular structures. J. Mol. Biol., Vol. 112, 1977, 535-542.

[BMUT97] S. Brin, R. Motwani, J.D. Ulman, S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basked Data. Proc. ACM International Conference on Management of Data (SIGMOD), Tucson, AZ, USA, 1997.

[BN05] J. Bleiholder, F. Naumann. Declarative Data Fusion - Syntax, Semantics, and Implementation. Proc. of the International Conference on Advances in Databases and Information Systems (ADBIS), Tallin, Estonia, 2005, 58-73.

[BN06] J. Bleiholder, F. Naumann. Conflict Handling Strategies in an Integrated Information System. In WWW Workshop in Information Integration on the Web (IIWeb), Edinburgh, UK, 2006.

[BO04] A.D. Baxevanis, B.F.F. Ouellette. Bioinformatics: A Practical Guide to the Analysis of Genes & Proteins. John Wiley & Sons, ISBN 0471 478784, 2004.

[BP01] S.D. Bay, M.J. Pazzani. Detecting group differences: Mining contrast sets. Data Mining and Knowledge Discovery, Vol. 5, *No. 3*, 2001, 213-246.

[Bre99] S.E. Brenner. Errors in genome annotation. Trends in Genetics, Vol. 15, *No. 4*, 1999, 132-133.

[BSVW04] T. Brijs, G. Swinnen, K. Vanhoof, G. Wets. Building an Association Rules Framework to Improve Product Assortment Decisions. Data Mining and Knowledge Discovery, Vol. 8, *No. 1*, 2004, 7-23.

[Bur99] C Burks. Molecular Biology Database List. Nucleic Acid Research, Vol. 27, 1999, 1-9, doi:10.1093/nar/27.1.1.

[CEK+03] J. Cheung, X. Estivill, R. Khaja, J.R. MacDonald, K. Lau, L.-C- Tsui, S.W. Scherer. Genome-wide detection of segmental duplications and potential assembly errors in the human genome sequence. Genome Biology, Vol. 4, 2003, 25.

[CGGM03] S. Chaudhuri, K. Ganjam, V. Ganti, R. Motwani. Robust and Efficient Fuzzy Match for Online Data Cleansing. Proc. ACM International Conference on Management of Data (SIGMOD), San Diego, CA, USA, 2003, 313-324.

[CG97] S. Chawathe, H. Garcia-Molina. Meaningful change detection in structured data. Proc. ACM International Conference on Management of Data (SIGMOD), Tucson, AZ, USA, 1997.

[CH01] H.-H. Chou, M.H. Holmes. DNA sequence quality trimming and vector removal. Bioinformatics, Vol. 17, *No. 12*, 2001, 1093-1104.

[Cla93] J.M. Claverie. Detecting frame shifts by amino acid sequence comparison. Journal Molecular Biology, Vol. 234, *No. 4*, 1993, 1140–1157.

[CM05] J. Chomicki, J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. Information and Computation, Vol. 197, *No. 1/2*, 2005, 90-121.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Vol. 13, *No. 6*, 1970, 377-387.

[CTX+04] G. Cong, A.K.H. Tung, X. Xu, F. Pan, J. Yang. FARMER: finding interesting rule groups in microarray datasets. Proc. ACM International Conference on Management of Data (SIGMOD), Paris, France, 2004, 143–154.

[CWO+04] S.S. Chow, C.O. Wilke, C. Ofria, R.E. Lenski, C. Adami. Adaptive Radiation from Resource Competition in Digital Organisms. Science, Vol. 305, *Issue 5680*, 2004, 84-86.

[CZW98] Y. Chen, Q. Zhu, N. Wang. Query processing with quality control in the World Wide Web. World Wide Web, Vol. 1, *No. 4*, 1998, 241-255.

[DG02] C. Dennis, R. Gallagher (Eds.). The Human Genome. Palgrave Macmillan, 2002.

[DLW84] P. Dadam, V.Y. Lum, H.-D. Werner. Integration of Time Versions into a Relational Database System. In Proc. of 10th International Conference on Very Large Data Bases, Singapore, 1984, 509-522.

[Doo87] R.F. Doolittle. Of URFs and ORFs – A Primer on How to Analyze Derived Amino Acid Sequences. University Science Books, 1987.

[DV01] D. Devos, A. Valencia. Intrinsic errors in genome annotation. Trends in Genetics, Vol. 17, *No. 8*, 2001, 429-431.

[EBR+01] S. M. Embury, S. M. Brand, J. S. Robinson, I. Sutherland, F. A. Bisby, W. A. Gray, A. C. Jones, R. J. White. Adapting integrity enforcement techniques for data reconciliation. Information Systems, Vol. 26, 2001, 657-689.

[EIV07] A.K. Elmagarmid, P.G. Ipeirotis, V.S. Verykios. Duplicate Record Detection: A Survey. IEEE Transations on knowledge and Data Engineering (TKDE), Vol. 19, *No. 1*, 2007, 1-16.

[EN00] R. Elmasri, S.B. Navathe. Fundamentals of Database Systems. Addison Wesley Pub Co., ISBN 0201542633, 2000.

[Fay98] U. Fayyad. Mining Database: Towards Algorithms for Knowledge Discovery. IEEE Technical Bulletin Data Engineering, Vol. 21, *No. 1*, 1998.

[FGM+03] F. Meyer, A. Goesmann, A.C. McHardy, D. Bartels, T. Bekel, J. Clausen, J. Kalinowski, B. Linke, O. Rupp, R. Giegerich, A. Pühler. GenDB—an open source genome annotation system for prokaryote genomes. Nucleic Acids Research, Vol. 31, *No. 8*, 2003, 2187-2195.

[FH76] P. Fellegi, D. Holt. A Systematic Approach to Automatic Edit and Imputation. Journal of the American Statistical Association, Vol. 71, 1976, 17-35.

[FQ95] G.A. Fichant, Y. Quentin. A frameshift error detection algorithm for DNA sequencing projects. Nucleic Acids Research, Vol. 23, *No. 15*, 1995, 2900–2908.

[FLMC01] W. Fan, H. Lu, S.E. Madnick, D. Cheung. Discovering and reconciling value conflicts for numerical data integration. Information Systems, Vol. 26, 2001, 635-656.

[GABF00] R. Guigó, P. Agarwal, M. Burset, J.W. Fickett. An Assessment of Gene Prediction Accuracy in Large DNA Sequences. Genome Research, Vol. 10, 2000, 1631-1642.

[Gal07] M.Y. Galperin. The Molecular Biology Database Collection: 2007 update. Nucleic Acids Research, Vol. 35, 2007, D3-D4, doi:10.1093/nar/gkl1008.

[GENB] GenBank Growth. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[GFSS00] H. Galhardas, D. Florescu, D. Shasha, E. Simon. AJAX: An extensible data cleaning tool. Proc. ACM International Conference on Management of Data (SIGMOD), Dallas, TX, USA, 2000.

[GFS+01] H. Galhardas, D. Florescu, D. Shasha, E. Simon, C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. Proc. Int. Conf. On Very Large Data Bases (VLDB), Roma, Italy, 2001.

[GJ79] M.R. Garey, D. S. Johnson. Computers and intractability. A guide to the theory of NP-completeness. W.H. Freeman and Co., San Francisco, CA, USA, 1979.

[GS01] G. Governatori, A. Stranieri. Towards the Application of Association Rules for Defeasible Rules Discovery. In Jurix 2001, Amsterdam, 2001, 63-75.

[GSS04] P. Gajer, M. Schatz, S.L. Salzberg. Automated correction of genome sequence errors. Nucleic Acids Research, Vol. 32, *No. 2*, 2004, 562-569.

[Gus97] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computer Biology. Cambridge University Press, ISBN 0-521-58519-8, 1997.

[Ham50] R. W. Hamming. Error-detecting and error-correcting codes. Bell System Technical Journal, Vol. 29, *No. 2*, 1950, 147-160.

[HBB+02] T. Hubbard, D. Barker, E. Birney, G. Cameron, Y. Chen, L. Clark, T. Cox, J. Cuff, V. Curwen, T. Down, R. Durbin, E. Eyras, J. Gilbert, M. Hammond, L. Huminiecki, A. Kasprzyk, H. Leh-

vaslaiho, P. Lijnzaad, C. Melsopp, E. Mongin, R. Pettett, M. Pocock, S. Potter, A. Rust, E. Schmidt, S. Searle, G. Slater, J. Smith, W. Spooner, A. Stabenau, J. Stalker, E. Stupka, A. Ureta-Vidal, I. Vastrik, M. Clamp. The Ensembl genome database project. Nucleic Acids Research, Vol. 30, *No. 1*, 2002, 38-41.

[Hen02] S. Hensley, Death of Pfizer's 'Youth Pill' Illustrates Drugmakers Woes. The Wall Street Journal online, May 2, 2002.

[HGN00] J. Hipp, U. Güntzer, G. Nakhaeizadeh. Algorithms for association rule mining — A general survey and comparison. ACM SIGKDD Explorations Newsletter, Vol. 2, *Issue 1*, 2000, 58-64.

[HGP+04] K.G. Herbert, N.H. Gehani, W.H. Piel, J.T.-L. Wang, C.H. Wu. BIO-AJAX: An Extensible Framework for Biological Data Cleaning. ACM SIGMOD Record, Vol. 33, *No. 2*, 2004, 51-57.

[HH04] T. Hrycej, J. Hipp. Outlier Detection by Rareness Assumption. Informatik 2004 Workshop Datenqualität – Ein zumeist unterschätzter Erfolgsfaktor, Ulm, Germany, 2004.

[Hin02] H. Hinrichs. Datenqualitätsmanagement in Data Warehouse-Systemen. Carl von Ossietzky Universität Oldenburg, Germany, 2002 (in german).

[HKK00] E. Han, G. Karypis, V. Kumar. Scalable Parallel Data Mining for Associatin Rules. IEEE Transations on knowledge and Data Engineering (TKDE), Vol. 12, *No. 2*, 2000.

[HKPT98] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen. Efficient Discovery of Functional and Approximate Dependencies Using Partitions. Proc. of the Int. Conf. on Data Engineering (ICDE), 1998, 392-401.

[HMS01] D. Hand, H. Mannila, P. Smyth. Principles of Data Mining. The MIT Press, ISBN 0-262-08290-X, 2001.

[HPY00] J. Han, J. Pei, Y. Yin. Mining Frequent Patterns without Candidate Generation. Proc. ACM International Conference on Management of Data (SIGMOD), Dallas, TX, USA, 2000.

[HRC+03] S. Hoon, K.K. Ratnapu, J. Chia, B. Kumarasamy, X. Juguang, M. Clamp, A. Stabenau, S. Potter, L. Clarke, E. Stupka. Biopipe: A Flexible Framework for Protocol-Based Bioinformatics Analysis. Genome Research, Vol. 13, 2003, 1904–1915.

[HS95] M.A. Hernandez, S.J. Stolfo. The merge/purge problem for large databases. Proc. ACM International Conference on Management of Data (SIGMOD), 1995.

[IHGSC04] International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. Nature, Vol. 431, *No. 7011*, 2004, 931-945.

[INSDC] International Nucleotide Sequence Database Collaboration, http://www.insdc.org.

[ITA+03] I. Iliopoulos, S. Tsoka, M.A. Andrade, A.J. Enright, M. Carroll, P. Poullet, V. Promponas, T. Liakopoulos, G. Palaios, C. Pasquier, S. Hamodrakas, J. Tamames, A.T. Yagnik, A. Tramontano, D. Devos, C. Blaschke, A. Valencia, D. Brett, D. Martin, C. Leroy, I. Rigoutsos, C. Sander, C.A. Ouzounis. Evaluation of annotation strategies using an entire genome sequence. Bioinformatics, Vol. 19, *No. 6*, 2003, 717-726.

[Jur88] J.M. Juran. Juran on Planing for Quality. The Free Press, New York, 1988.

[JV97] M. Jarke, Y. Vassiliou. Data Warehouse QualityDesign: A review of the DWQ projects. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 1997.

[KAA+07] T. Kulikova, R. Akhtar, P. Aldebert, N. Althorpe, M. Andersson, A. Baldwin, K. Bates, S. Bhattacharyya, L. Bower, P. Browne, M. Castro, G. Cochrane, K.Duggan, R. Eberhardt, N. Faruque, G. Hoad, C. Kanz, C. Lee, R. Leinonen, Q. Lin, V. Lombard, R. Lopez, D. Lorenc, H. McWilliam, G. Mukherjee, F. Nardone, M.P. Garcia-Pastor, S. Plaister, S. Sobhany, P. Stoehr, R. Vaughan, D. Wu, W. Zhu, R. Apweiler. EMBL Nucleotide Sequence Database in 2006. Nucleic Acids Research, Vol. 35, 2007, D16-D20.

[KCGS93] W. Kim, I. Choi, S. Gala, M. Scheevel. On resolving schematic heterogeneity in multidatabase systems. Distributed and Parallel Databases, Springer, Vol. 1, *No. 3*, 1993, 251-279.

[KCH+03] W. Kim, B.-J. Choi, E.-K. Hong, S.-K. Kim, D. Lee. A Taxonomy of Dirty Data. Data Mining and Knowledge Discovery, Vol. 7, *No. 1*, 2003, 81-99.

[KH95] K. Koperski, J. Han. Discovery of spatial association rules in geographic information databases. Lecture Notes In Computer Science, 951, 1995, 47-66.

[KHCV02] A. Krause, S.A. Haas, E. Coward, M. Vingron. SYSTERS, GeneNest, SpliceNest: exploring sequence space from genome to protein. Nucleic Acids Research, Vol. 30, *No. 1*, 2002, 299-300.

[KL05] N. Kaplan, M. Linial. Automatic detection of false annotations via binary roperty Clustering. BMC Bioinformatics, Vol. 6, *No. 46*, 2005, doi:10.1186/1471-2105-6-46.

[KLK+04] J.L.Y.Koh, M.L. Lee, A.M. Khan, P.T.J. Tan, V. Brusic. Duplicate Detection in Biological Data using Association Rule Mining. Second European Workshop on Data Mining and Text Mining for Bioinformatics, Pisa, Italy, 2004.

[KS91] W. Kim, J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. IEEE Computer, Vol. 24, *No. 12*, 1991, 12-18.

[Kus02] A. Kusiak. A Data Mining Approach for Generation of Control Signatures. Journal of Manufacturing Science and Engineering, Vol. 124, *Issue 4*, 2002, 923-926.

[LAB+05] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao. Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows, 2005.

[Len02] M. Lenzerini. Data Integration: A Theoretical Perspective. Proc. ACM Symposium on Principles of Database Systems (PODS), Madison, Wisconsin, USA, 2002.

[LD60] A.H. Land, A.G. Doig. An automatic method of solving discrete programming problems. In Econometrica ,Vol. 28, 1960, 497-520.

[Lev65] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR, Vol. 163, *No. 4*, 1965, 845-848 (Russian). English translation in Soviet Physics Doklady, Vol. 10, *No. 8*, 1966, 707-710.

[LG96] W. J. Labio, H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. Proc. Int. Conf. On Very Large Data Bases (VLDB), Bombay, India, 1996, 63-74.

[LHM98] B. Liu, W. Hsu, Y. Ma. Integrating Classification and Association Rule Mining. Proceedings of the ACM SIGKDD, New York, 1998.

[Lin03] M. Linial. How incorrect annotations evolve – the case of short ORFs. Trends in Biotechnology, Vol. 21, *No. 7*, 2003, 298-300.

[LLL00] Mong Li Lee, Tok Wang Ling, Wai Lup Low. IntelliClean: A knowledge-based intelligent data cleaner. Proceedings of the ACM SIGKDD, Boston, MA, USA, 2000.

[LLL01] Wai Lup Low, Mong Li Lee, Tok Wang Ling. A knowledge-based approach for duplicate elimination in data cleaning. Information Systems, Vol. 26, 2001, 585-606.

[Los01] D. Loshin. Enterprise Knowledge Management – The Data Quality Approach. Morgan Kaufmann Publishers, Academic Press, ISBN 0-12-455840-2, 2001.

[LR87] R.J.A. Little, D.B. Rubin. Statistical Analysis with Missing Data. Wiley & Sons, ISBN 0-471-80254-9, 1987.

[LS02] J. Long, C Seko. A new method for database quality evaluation at the Canadian Institue for Health Information. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2002, 238-250.

[ME97] A.E. Monge, C.P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database tuples. Proceedings of the SIGMOD 1997 workshop on data mining and knowledge discovery, 1997.

[MF03] H. Müller, J.-C. Freytag. Problems, Methods, and Challenges in Comprehensive Data Cleansing. HUB-IB-164, Humboldt University Berlin, 2003.

[MFL06] H. Müller, J.-C. Freytag, U. Leser. Describing Differences between Databases. ACM 15th Conference on Information and Knowledge Management (CIKM), Arlington, VA, USA, 2006.

[MLF04] H. Müller, U. Leser, J.-C. Freytag. Mining for Patterns in Contradictory Data, Proc. SIGMOD Int. Workshop on Information Quality for Information Systems (IQIS'04), Paris, France, 2004.

[MLF06a] H. Müller, U. Leser, J.-C. Freytag. Classification of Contradiction Patterns. 30th Annual Conference of the German Classification Society (GfKl), Berlin, Germany, 2006.

[MLF06b] H. Müller, U. Leser, J.-C. Freytag. On the Distance of Databases. HUB-IB-199, Humboldt University Berlin, 2006.

[MM00] J.I. Maletic, A. Marcus. Data Cleansing: Beyond Integrity Analysis. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2000.

[MMN05] M. Mielke, H. Müller, F. Naumann. Ein Data-Quality-Wettbewerb. Datenbank-Spektrum, *Heft 14*, 2005, 34-37.

[MML01] A. Marcus, J.I. Maletic, K.I. Lin. Ordinal Association Rules for Error Identification in Data Sets. Proc. of the 10th ACM International Conference on Information and Knowledge Management (CIKM), Atlanta, GA, USA, 2001, 589-591.

[MNF03] H. Müller, F. Naumann, J.-C. Freytag. Data Quality in Genome Databases. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2003.

[Mot89] A. Motro. Integrity = Validity + Completeness. ACM Transactions on Database Systems (TODS), Vol. 14, *No. 4*, 1989, 480-502.

[MT99] E. Mayol, E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. ER Workshops, 1999, 62-73.

[MTV97] H. Mannila, H. Toivonen, A.I. Verkamo. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery, Vol. 1, *No. 3*, 1997, 259-289.

[Mül03] H. Müller. Semantic Data Cleaning in Genome Databases. Proc. of the VLDB 2003 PhD Workshop, Berlin, Germany, 2003.

[MWBL05] H. Müller, M. Weis, J. Bleiholder, U. Leser. Erkennen und Bereinigen von Datenfehlern in naturwissenschaftlichen Daten. Datenbank-Spektrum, *Heft 15*, 2005, 26-35 (in german).

[MWO] Merriam-Webster Online, http://www.m-w.com.

[Nau02] F. Naumann. Quality-driven Query Answering for Integrated Information Systems. Lecture Notes in Computer Science, Springer Verlag, 2002.

[NH02] F. Naumann, M. Häussler. Declarative Data Merging with Conflict Resolution. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2002.

[NÖK+04] .A. Nascimento, M.T. Özsu, D. Kossmann. R.J. Miller, J.A. Blakeley, K.B. Schiefer. Proceedings of the Thirtieth International Conference on Very Large Data Bases. Morgan Kaufman, 2004.

[NYC06] H. Ning, H. Yuan, S. Chen. Temporal Association Rules in Mining Method. First International Multi-Symposiums on Computer and Computational Sciences, 2006, 739-742.

[OES06] C. Ordonez, N. Ezquerra, C.A. Santana. Constraining and summarizing association rules in medical data. Knowledge and Information Systems, Vol. 9, *Issue 3*, 2006, 259 - 283.

[Ols03] J. E. Olson. Data Quality – The Accuracy Dimension. Morgan Kaufmann Publishers, ISBN 1-55860-891-5, 2003.

[ORH05] P. Oliveira, F. Rodrigues, P. Henriques. A Formal Definition of Data Quality Problems. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2005.

[OSB00] C. Ordonez, C.A. Santana, L. de Braal. Discovering interesting association rules in medical data. In ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD 2000, 2000), 78-85.

[OSGT06] K. Okubo, H. Sugawara, T. Gojobori, Y. Tateno. DDBJ in preparation for overview of research activities behind data submissions. Nucleic Acids Research, Vol. 34, 2006, D6–D9.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal. Discovering Frequent Closed Itemsets for Association Rules. Lecture Notes in Computer Science, Vol. 1540, 1999, 398-416.

[PCT+03] F. Pan, G. Cong, A.K.H. Tung, J Yang, M.J. Zaki. CARPENTER: finding closed patterns in long biological datasets. Proc. Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD), Washington DC, USA, 2003.

[PCY95] J. Park, M. Chen, P.S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. Proc. ACM International Conference on Management of Data (SIGMOD), 1995.

[PG02] F. Piontek, H. Groot. Healthcare informatics: Data quality, warehousing and mining applications. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2002, 256-260.

[PHBR04] H. Pospisil, A. Herrmann, R.H. Bortfeldt, J.G. Reich. EASED: Extended Alternatively Spliced EST Database. Nucleic Acids Research, Vol. 32, *Database issue*, 2004, D70-D74.

[PKS02] P.-N. Tan, V. Kumar, J. Srivastava. Selecting the Right Interestingness Measure for Association Patterns. Proc. ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), 2002.

[PR92] J. Posfai, R.J. Roberts. Finding errors in DNA sequences. Proc. Natl. Acad. Sci. U S A., Vol. 89, *No. 10*, 1992 , 4698–4702.

[Pyl99] D. Pyle. Data preparation for Data Mining. Morgan Kaufmann Publishers, ISBN 1-55860-529-0, 1999.

[RD00] E. Rahm, H.H. Do. Data Cleaning: Problems and current approaches. IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 24, *No. 4*, 2000.

[Red96] T. Redman. Data Quality for the Information Age. Artech House, Boston, London, 1996.

[Red98] T. Redman. The Impact of Poor Data Quality on the Typical Enterprise. Communications of the ACM, Vol. 41, *No. 2*, 1998.

[RH01] V. Raman, J.M. Hellerstein. Potter's Wheel: An Interactive Framework for Data Transformation and Cleaning. Proc. Int. Conf. On Very Large Data Bases (VLDB), Roma, Italy, 2001.

[Ric98] P. Richterich. Estimation of Errors in „Raw" DNA Sequences: A Validation Study. Genome Research, Vol. 8, 1998, 251-259.

[RMT+04] K. Rother, H. Müller, S. Trissl, I. Koch, T. Steinke, R. Preissner, C. Frömmel, U. Leser. Columba: Multidimensional Data Integration of Protein Annotations. Int. Workshop on Data Integration in Life Sciences (DILS 2004), Leipzig, Germany, 2004.

[RTM+05] K. Rother, S. Trißl, H. Müller, T. Steinke, I. Koch, R. Preissner, C. Frömmel, U. Leser. Columba: An Integrated Database of Proteins, Structures, and Annotations. BMC Bioinformatics, Vol. 6, *No.1:81*, 2005.

[SCS00] K.-U. Sattler, S. Conrad, G. Saake. Adding Conflict Resolution Features to a Query Language for Database Federations. Proc. 3nd Int. Workshop on Engineering Federated Information Systems, Dublin, Ireland, 2000.

[SHM+99] C. Sapia, G. Höfling, M. Müller, C. Hausdorf, H. Stoyan, U. Grimmer. On Supporting the Data Warehouse Design by Data Mining Techniques. Proc. GI-Workshop Data Mining and Data Warehousing, 1999.

[SL90] A.P. Sheth, J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys, Vol. 22, *No. 3*, 1990, 183-236.

[Sla96] P. Slavik. A Tight Analysis of the Greedy Algorithm for Set Cover. ACM Symposium on Theory of Computing (STOC), Philadelphia, USA, 1996.

[SLW97] D.M. String, Y.W. Lee, R.Y. Wang. Data Quality in Context. Communications for the ACM, Vol. 40, *No. 5*, 1997.

[SPD92] S. Spaccapietra, C. Parent, Y. Dupont. Model independent assertions for integration of heterogeneous schemas. Journal on Very Large Data Bases, Vol. 1, *No. 1*, 1992, 81-126.

[SS01] K.-U. Sattler, E. Schallehn. A Data Preparation Framework based on a Multidatabase Language. International Database Engineering Applications Symposium (IDEAS), Grenoble, France, 2001.

[STR+03] T.E. Scheetz, N. Trivedi, C.A. Roberts, T. Kucaba, B. Berger, N.L. Robinson, C.L. Birkett, A.J. Gavin, B. O'Leary, T.A. Braun, M.F. Bonaldo, J.P. Robinson, V.C. Sheffield, M.B. Soares, T.L. Casavant. ESTprep: preprocessing cDNA sequence reads. Bioinformatics, Vol. 19, *No. 11*, 2003, 1318-1324.

[SV00] A. Salleb, C. Vrain. An Application of Association Rules Discovery to Geographic Information Systems. Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), Lyon, France, 2000.

[SW81] T.F. Smith, M.S. Watermann. Identification of common molecular subsequences. Journal of Molecular Biology, Vol. 147, 1981, 195-197.

[SWZ00] G. Shankaranarayanan, R.Y. Wang, M. Ziad. IP-MAP: Representing the Manufacture of an Information Product. Proc. Int. Conf. On Information Quality (ICIQ), Cambridge, MA, USA, 2000, 1-16.

[TB98] G.K. Tayi, D.P. Ballou. Examining Data Quality. Communications of the ACM, Vol. 41, *No. 2*, 1998.

[Toi96] H. Toivonen. Sampling Large Databases for Association Rules. Proc. Int. Conf. On Very Large Data Bases (VLDB), Bombay, India, 1996.

[TSC+04] T.A. Thanaraj, S. Stamm, F. Clark, J.-J. Riethoven, V. Le Texier, J. Muilua. ASD: the Alternative Splicing Database. Nucleic Acids Research, Vol. 32, *Database issue*, 2004, D64–D69.

[UNI07] The UNIPROT Consortium. The universal protein resource (UNIPROT). Nucleic Acid Research, Vol. 35, *Database Issue*, 2007, D193-197.

[VAM+01] J.C. Venter, M.D. Adams, E.W. Myers. The sequence of the human genome. Science, Vol. 291, *No. 5507*, 2001, 1304−1351.

[Vos91] G. Vossen. Data Models, Database Languages and Database Management Systems. Addison-Wesley Publishers, ISBN 0-201-41604-2, 1991.

[VVS+01] P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis, T. Sellis. ARKTOS: towards the modeling, design, control and execution of ETL processes. Information Systems, Vol. 26, 2001, 537-561.

[WBN03] G.I. Webb, S.M. Butler, D. Newlands. On detecting differences between groups. Proc. of the Ninth ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Washington, DC, USA, 2003.

[WC02] W. Winkler, B.-C. Chen. Extending the Fellegi-Holt Model of Statistical Data Editing. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, USA, 2002.

[Wei99] G. Weikum. Towards guaranteed quality and dependability of information systems. Proc. of the Conf. Datenbanksysteme im Büro, Technik und Wirtschaft (BTW), Freiburg, Germany, 1999, 379-409.

[WHP03] J. Wang, J. Han, J. Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Washington DC, USA, 2003, 236-245.

[Wij03] J. Wijsen. Condensed representation of database repairs for consistent query answering. In Proc. of the 9th Int. Conf. on Database Theory, Siena, Italy, 2003.

[Win99] W. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington DC, USA, 1999.

[WKA04] D. Wieser, E. Kretschmann, R. Apweiler. Filtering erroneous protein annotation. Bioinformatics, Vol. 20, *Suppl. 1*, 2004, i342-i347.

[WNB06] M. Weis, F. Naumann, F. Brosy. A Duplicate Detection Benchmark for XML (and Relational) Data. SIGMOD Workshop on Information Quality for Information Systems (IQIS), Chicago, IL, USA, 2006.

[WS96] R.Y. Wang, D.M. Strong. Beyond accuracy: What data quality means to data consumers. Journal of Management Information Systems, Vol. 12, *No. 4*, 1996, 5-34.

[XTK03] H. Xiong, P.-N. Tan, V. Kumar. Mining Strong Affinity Association Patterns in Data Sets with Skewed Support Distributions. In Proc. of IEEE Int. Conf. on Data Mining, Melbourne, FL, USA, 2003.

[ZH02] M.J. Zaki, C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In Proc. of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, 2002.

Erklärung

Ich erkläre hiermit, daß

- ich die vorliegende Dissertationsschrift „Describing Differences between Overlapping Databases“ selbständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Berlin, den 22. April 2008